MASTER EN SCIENCES INFORMATIQUES

Contribution à la spécification formelle des besoins en sécurité pour le logiciel de messagerie électronique X.400, EAN

Honet, Laurent

Award date:
1993

Awarding institution:
Universite de Namur

CONTRIBUTION A LA SPECIFICATION FORMELLE DES BESOINS EN SECURITE POUR LE LOGICIEL DE MESSAGERIE ELECTRONIQUE X.400, EAN.

Facultés Universitaires Notre-Dame de la Paix
INSTITUT D'INFORMATIQUE
Rue Grandgagnage, 21 - 5000 NAMUR
Tél.081/72.41.11

CONTRIBUTION A LA SPECIFICATION FORMELLE DES BESOINS EN SECURITE POUR LE LOGICIEL DE MESSAGERIE ELECTRONIQUE X.400, EAN.

HONET Laurent

Résumé

La mise en place de services de sécurité en messagerie électronique est exigée par un nombre sans cesse croissant d'utilisateurs.
Des services de sécurité ont été introduits dans les recommandations X.400 version 1988. L'objet de notre travail consiste en une spécification formelle de ceux-ci afin de les inclure aux recommandations X.400 version 1984. Cette spécification, exécutée en langage ALBERT, a pour but d'adapter le logiciel EAN à ces services et de définir une architecture des services de sécurité.

Mémoire de licence et maîtrise en Informatique
Septembre 1993
Promoteur : J. Ramaekers

Avant de commencer l'exposé du travail, je tiens à adresser mes plus vifs remerciements

à Monsieur le Professeur RAMAEKERS, pour l'accueil qu'il m'a réservé au sein de son groupe de recherche à l'Institut d'Informatique ainsi que pour les précieux conseils et l'intérêt qu'il a apporté à mon travail,

à Monsieur le Professeur DUBOIS, pour m'avoir proposé l'utilisation du langage ALBERT et pour ses précieux conseils méthodologiques quant à l'utilisation de celui-ci,

à Suchun WU, pour sa coopération et sa grande disponibilité,

à Philippes DU BOIS, pour son aide considérable et sa sympathie,

à Marc DERROITTE, pour son appui technique,

à Joël HUBIN, pour son aide,

à Monsieur Pierre BREUER, pour m'avoir accordé des facilités pour l'exécution de ce travail,

et à Nathalie, pour son soutien moral.

TABLE DES MATIERE.

Introduction générale.

Dans notre société moderne, l'informatique est présente dans des domaines de plus en plus variés, tels que les domaines privé, bancaire, médical, scientifique,... Et un nombre croissant d'utilisateurs ont souhaité interconnecter leurs ordinateurs afin de pouvoir communiquer à distance. Une application pratique pouvant en résulter est le courrier électronique, appelé aussi messagerie électronique. Ces utilisateurs ont exprimé avec force des besoins en services de sécurité. Il désirent, en effet, assurer au transfert de leurs messages la plus haute sécurité possible, tant au niveau de leur confidentialité qu'au niveau de leur intégrité.

La première version de recommandations X.400, définie en 1984 par l'organisme de Standardisation CCITT (X.400-84) ne répondait quasiment pas à ces besoins en services de sécurité. Par contre, la version revue en 1988 (X.400-88) par l'International Standard Organisation (ISO) introduisit de tels services, mais resta fort générale quant au mode de fonctionnement et d'utilisation de ceux-ci.

Le logiciel EAN dont nous disposions pour ce travail est basé sur les recommandations X.400-84. Puisque celui-ci ne proposent pas de services de sécurité et que notre but est justement d'en adjoindre au logiciel, nous avons repris tous les concepts relatifs à ces services dans X.400-88. Etant donné que ces deux versions présentent des différences importantes en ce qui concerne certains concepts, nous avons préféré choisir de respécifier les services de sécurité de X.400-88 et de les inclure au monde de X.400-84. C'est le langage ALBERT (**A**gent-oriented **L**anguage for **B**uilding and **E**liciting **R**equirements for real-**T**ime systems) que nous avons choisi d'utiliser pour cette respécification, laquelle nous permettra de définir une architecture des services de sécurité.

Les chapitres contenus dans ce mémoire sont disposés d'une façon telle qu'ils permettent au lecteur de s'adapter à notre spécification des services de sécurité en langage ALBERT.

Après une introduction rappelant les concepts principaux de X.400 (chapitre I), nous initierons le lecteur au langage ALBERT (chapitre II). Seront ensuite reformulés en ALBERT les besoins décrits dans X.400-84, en prenant comme cas le logiciel EAN (chapitre III). Afin d'évoluer vers la spécification d'un serveur de sécurité (chapitre VI), nous exposerons les principaux concepts de la cryptographie (chapitre IV) et nous décrirons les services de sécurité présentés dans X.400-88 (chapitre V). Enfin, nous terminerons notre travail par une série de conclusions.

Chapitre I:

Introduction à la recommandation X.400.

Début des années 80, des ressources importantes ont été mobilisées par des organismes de standardisation internationaux en vue de la description d'un modèle architectural de référence décrivant l'interconnexion de systèmes dits ouverts. Le modèle OSI (Open System Interconnection) défini par l'ISO (International Standard Organization) est le résultat de ces travaux. Il s'agit d'un modèle conceptuel pour l'échange d'informations entre terminaux, ordinateurs, personnes, réseaux, processus, etc,...qui sont mutuellement "ouverts" l'un à l'autre par le biais de leur utilisation commune de standards existants. Ce modèle énonce les fonctions à assurer dans le cadre le plus général, et les regroupe en sept couches fonctionnelles homogènes. Ces couches fonctionnelles se superposent les unes aux autres. Chaque couche s'appuie sur les services fournis par la couche adjacente "inférieure", et apporte une valeur ajoutée à ces services afin de fournir elle-même au niveau adjacent qui lui est "supérieur" un service plus complet. Une couche fonctionnelle dialogue avec son homologue en site distant sur base de règles (ou protocoles) décrites formellement dans des documents normatifs (ISO,...), de standards de fait,... Une couche appuyant son fonctionnement sur les services de la couche inférieure indépendamment de la façon dont cette couche inférieure travaille, on pourra remplacer cette couche par une autre définie selon une norme différente pour autant que les services rendus soient identiques. La couche de niveau le plus élevé étant le "client" final du système de communications, c.-à-d. l'application elle-même, pourra donc profiter des services de communication indépendamment de la façon dont ceux-ci sont rendus. L'utilisation des normes conformantes à OSI garantit donc la transparence de l'application vis-à-vis de moyens de communication. Concrètement parlant, on pourra développer une application consommatrice de communications sans savoir à priori quels types de réseaux seront utilisés.

Une classe de ces applications est la classe des systèmes de messagerie électronique (Message Handling System, ou MHS). En 1984, le CCITT proposa la première version des recommandations X.400, qui a ensuite servie de base au travail appelé MOTIS (Message Oriented Text Interchange System) à l'ISO. En 1988, le CCITT modifia X.400 pour le rendre cohérent avec MOTIS. C'est ainsi que MOTIS/X.400 est rapidement devenu un vrai standard de la messagerie électronique. Il existe donc deux versions de X.400: la version 1984 et la version 1988 que nous appellerons tout au long de ce document X.400-84 et X.400-88.

La recommandation X.400 est en réalité constituée de toute une série de documents dont chacun décrit un aspect bien précis de X.400. Le tableau 1.1. montre la liste de ces documents et leur référence pour les deux versions.

1984	1988	Sujet
X.400	X.400	Vue générale du système et éléments de service
X.401		Eléments de service de base et facilités optionnelles d'utilisateur
	X.402	Architecture générale
	X.403	Test de conformité
	X.407	Conventions de définition de service abstrait
X.408	X.408	Règles de conversion de type d'information encodée (EIT)
X.409	X.208	Notation de syntaxe de présentation (Langage ASN.1)
	X.209	
	X.410	"Remote operations" et "Reliable transfer server"
X.411	X.411	Système de transfert de messages: services abstraits et procédures
	X.413	Message Store: services abstraits
	X.419	Spécifications de protocoles (Protocoles P1,P3 et P7)
X.420	X.420	Système de messagerie interpersonnelle
X.430		Protocoles d'accès pour terminaux Télétex

Tableau 1.1.: Relations entre les recommandations X.400-84 et X.400-88.

Nous allons d'abord décrire les aspects de X.400-84, suivi d'une description des nouveaux concepts introduits par X.400-88.

I.1. Introduction générale à la recommandation X.400-84.

I.1.1. Architecture générale.

X.400 est basé sur deux composants principaux: le MTA (Message Transfert Agent) et l'UA (User Agent). Le MTA est un dispositif qui reçoit un message en provenance d'un UA ou d'un autre MTA et le réexpédie vers un autre UA ou un autre MTA. L'UA est une interface qui interagit directement avec l'utilisateur et fournit à celui-ci les fonctions qui lui permettent de préparer, d'envoyer et de recevoir des messages.

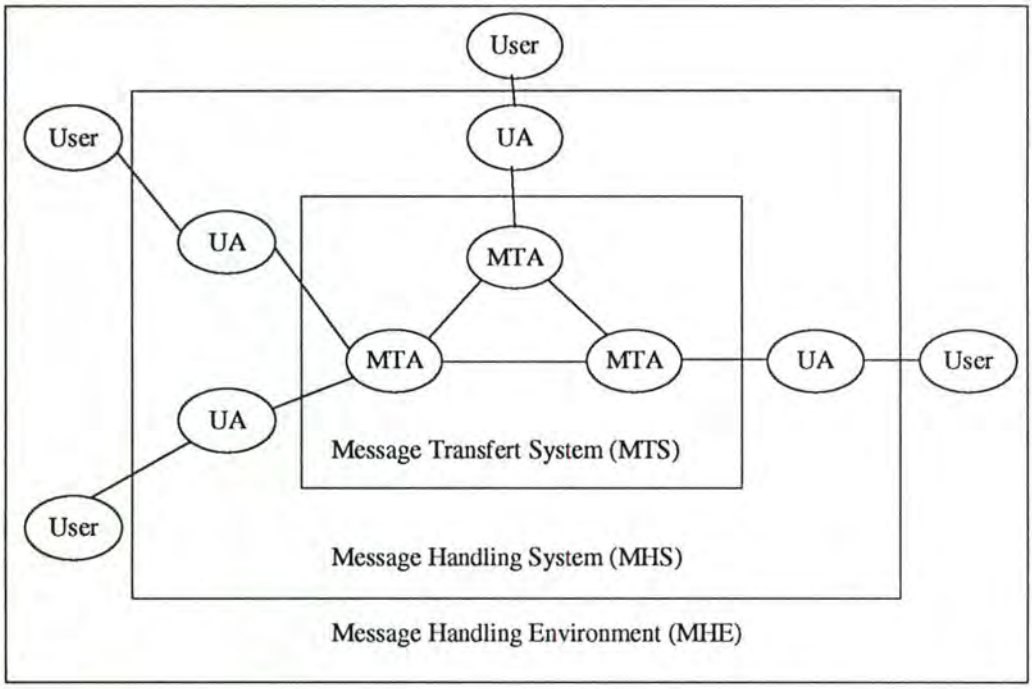

Figure 1.1. : Modèle fonctionnel du service de messagerie X.400.

La figure 1.1. montre le modèle fonctionnel de X.400. On peut diviser le service de messagerie X.400 en trois domaines:

a) Le Message Tranfert System, ou Système de Transert de Messages (MTS), formé par les MTAs. Le protocole utilisé entre les MTAs est appelé le protocole P1.

b) Le Message Handling System, ou Système de Traitement de Messages (MHS), formé par le MTS et les UAs. Pour le contenu, c.-à-d. le message échangé entre les UAs, le CCITT n'a défini qu'un seul type: le message interpersonnel (Interpersonal Message ou IPM). Ce message est régi par le protocole P2 et est appelé **message P2.** Dans ce cas, les UAs communiquant entre eux forment une classe bien particulière et sont appelés IPMUAs (Interpersonal Message UAs).

c) Le Message Handling Environment, ou Environnement de Traitement de Messages (MHE), formé par les MHS et les utilisateurs. Le protocole utilisé entre les utilisateurs n'est pas formalisé et ne fait pas partie de X.400.

X.400 fonctionne selon le principe du "Store-and-Forward". Cela signifie que l'établissement préalable d'une connexion réseau entre l'UA expéditeur et l'UA destinataire n'est pas nécessaire.

Le transfert de type "Store-and-Forward" (S/F) a été longtemps utilisé aux niveaux paquet et réseau comme moyen d'établissement de circuits virtuels et de transports de datagrammes. Cette technique est aussi largement utilisée, au niveau application par les

systèmes de messagerie électronique. Le transfert orienté message de type "S/F" (fig. 1.2.) est un transfert de message depuis un expéditeur jusqu'à un destinataire, en passant le message d'un MTA au suivant jusqu'au moment où le message arrive à destination. Dans le contexte de la série de Recommandations X.400, le message utilisé dans le MTS est appelé **message P1**. Il est constitué d'une enveloppe et d'un contenu, est régi par le protocole P1, et peut être adressé à plusieurs destinataires. L'unité qui est responsable du processus "Store-and-Forward" sur les messages P1 est le MTA. Deux MTAs sont en connexion "message" s'il existe un chemin "S/F" quelconque entre eux sans qu'il n'existe nécessairement une connexion disponible entre eux. C'est ainsi que nous parlerons de connexion "message" plutôt que de connexion réseau: par contre, une paire de MTAs peut établir une connexion réseau s'ils sont adjacents. Le MTA qui établit une connexion vers un MTA adjacent dans le but de transmettre un message est appelé MTA expéditeur, tandis que que l'autre s'appelle le MTA destinataire. Le principe "Store-and-Forward" fournit au MTS les caractéristiques suivantes:

-il n'est pas nécessaire d'établir une connexion de bout en bout pour transférer un message,
-un délai indéterminé (mais fini) peut se produire avant que le message soit transmis vers le MTA adjacent,
-plusieurs types de support de transmission (téléphone, Ethernet, lignes louées) peuvent être utilisés par le MTS,
-les MTAs adjacents peuvent utiliser n'importe quel protocole efficace (ISO Transport, TCP/IP, ...) pour transférer les messages.

Plusieurs systèmes de courrier électronique utilisent la technique "Store-and-Forward" pour transférer du courrier: les systèmes basés sur UUCP, basés sur SMTP, et les systèmes basés sur la série de recommandations X.400 tel que EAN.

Pour X.400, le transfert d'un message se fait selon les étapes suivantes (fig. 1.2.):

- **Préparation** d'un message
  L'UA de l'expéditeur assiste celui-ci dans la préparation du message.
- **Soumission** du message
  L'UA de l'expéditeur soumet le message à son MTA et lui fournit les informations nécessaires à la création de l'enveloppe.
- **Relais** du message
  Le MTS se charge du routage en se basant sur les informations de l'enveloppe. Suivant le principe du "Store-and-Forward", le message accompagné de son enveloppe passe de MTA en MTA pour arriver au MTA destinataire (MTA sur lequel le UA destinataire est connecté). Chaque MTA intermédiaire recevant un message le sauve temporairement ("Store") en attendant de le renvoyer au MTA ou à l'UA suivant ("Forward").
- **Livraison** du message
  Le MTA destinataire délivre le message à l'UA du récepteur si celui-ci est en état de le recevoir. Sinon, le MTA destinataire garde provisoirement le message pour le délivrer plus tard, pour autant que sa capacité de stockage le lui permet.
- **Réception** du message
  L'UA du destinataire confie le message au destinataire.

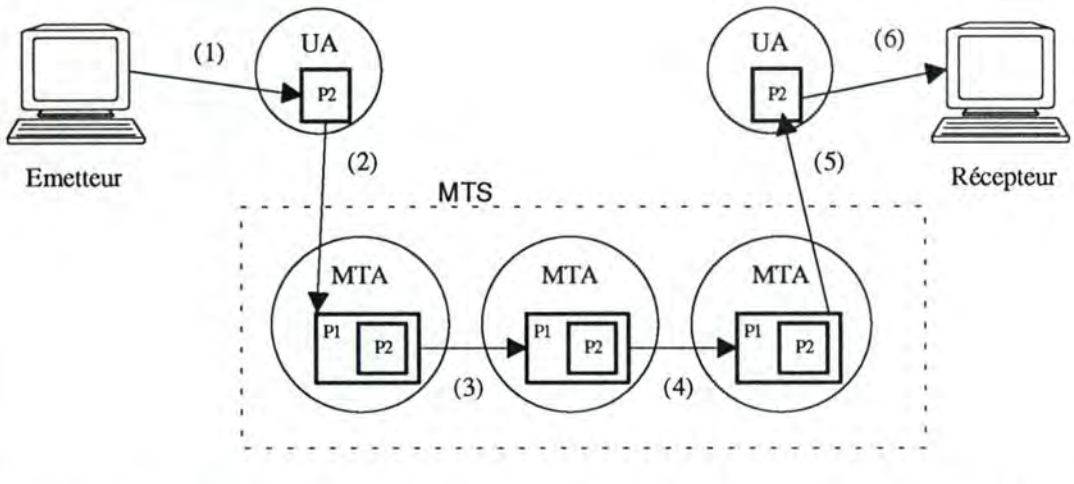

Figure 1.2.: Transfert de message interpersonnel par le service de messagerie électronique X.400: (1) Préparation du message; (2) Soumission du message au MTS; (3) et (4) Relais du message à travers le MTS; (5) Livraison du message au UA du récepteur; (6) Réception du message par le récepteur.

Remarques:

1°) Le système comprend des sécurités pour éviter le bouclage des messages dans le MTS (c.-à-d. qu'un message ne peut revenir sur un MTA par lequel il est déjà passé).
2°) Les MTAs peuvent transporter n'importe quels types de messages, même des messages binaires quelconques.

I.1.2. Les domaines de gestion (Management Domain).

Pour faciliter la gestion et l'administration du MHS, le MTS et les UAs sont partitionnés en **domaines de gestion** (**Management Domain** ou **MD**). Par conséquent, un MD est une partie du MHS contrôlée par un organisme unique. Chaque MD doit au moins contenir un MTA. Par contre, un MD peut contenir un nombre quelconque d'UAs. L'objectif des MDs est de déléguer la responsabilité de l'organisation et de la gestion du système aux autorités capables d'assumer cette tâche. Chaque domaine est responsable de sa gestion interne, en particulier du routage approprié des messages et de leur livraison. Un message qui entre dans un domaine reste sous sa responsabilité jusqu'au moment où il est délivré dans le domaine ou relayé à un domaine extérieur.

Si l'organisme gérant le MD est une administration publique, alors on appelle le MD **ADministration Management Domain** (**ADMD**). S'il est privé , on l'appelle **PRivate administration Management Domain** (**PRMD**).

La découpe en MDs fait les hypothèses suivantes ([X400], section 2.3.2.4.):

-un MD doit toujours être contenu entièrement dans un seul pays,
-un PRMD ne peut jamais servir d'intermédiaire entre deux ADMD,
-un PRMD ne peut être relié directement à un ADMD d'un autre pays: il doit passer par l'intermédiaire d'un ADMD du pays du PRMD.

Un des rôles de l'ADMD est d'agir comme autorité de nomination (naming authority) vis-à-vis des PRMDs qui sont sous sa responsabilité. Cela permet à l'ADMD de superviser l'attribution des noms aux PRMDs de son domaine c.-à-d. des noms de haut niveau; la responsabilité pour l'attribution des noms à l'intérieur du PRMD étant déléguée au PRMD.

I.1.3. L'adressage dans X.400-84.

I.1.3.1. O/R names et O/R address.

Chaque utilisateur est identifié par un nom appelé **O/R name** (**Originator/Recipient name**). Comme chaque utilisateur est associé à un et un seul UA, chaque O/R name identifie aussi bien un utilisateur que son UA. Actuellement, un O/R name n'est rien d'autre qu'une liste d'un ou plusieurs **attributs** (un attribut est une caractéristique quelconque de l'UA).

Lorsque les attributs de l'O/R name spécifient où se trouve l'UA de l'utilisateur, on dit que cet O/R name est une **O/R address**. Cette adresse permet de localiser précisément l'UA d'un utilisateur. L'information contenue dans l'O/R address du destinataire est utilisée pour déterminer le routage dans le MTS, c.-à-d. le chemin à suivre par ce message pour être délivré. Lorsque l'O/R name d'un destinataire d'un message soumis au MTS n'est pas une O/R address, cet O/R name doit d'abord être converti en une O/R address pour que le message puisse être délivré correctement. Ceci nécessite l'utilisation d'un service d'annuaire ("**directory service**") contenant pour chaque utilisateur la liste de ses attributs, et en particulier une O/R address. Ce service d'annuaire n'était pas normalisé lors de l'élaboration de la version 1984 de X.400. Depuis lors, des travaux de normalisation ont conduit à l'élaboration de la norme X.500 (ISO Draft 9594-1, en 1988).

I.1.3.2. Les attributs utilisés pour représenter les O/R names.

X.400 reconnaît un certain nombre d'attributs standardisés (**standard attributes**) et des attributs non standardisés, définis uniquement dans le domaine de l'utilisateur (**domain-defined attributes**, i.e. **DDA**s). Parmi les attributs standardisés, on trouve les attributs personnels (nom personnel, les initiales,...), géographiques (nom de rue et numéro, ville, région, pays,...), organisationnels (nom de l'organisation, de l'unité organisationnelle, position ou rôle), architecturaux (adresse X.121, identifiant unique de l'UA, nom de L'ADMD, PRMD). Les DDAs ont été définis essentiellement pour permettre l'interconnexion X.400 des systèmes existant qui utilisaient déjà certains attributs non standardisés ou d'autres méthodes d'adressage (adressage RFC822 dans la version UBC1 de EAN).

X.400 définit quatre formes de construction de l'O/R name. Chaque forme spécifie un ensemble d'attributs qui permet d'identifier un utilisateur unique dans le MTS. Voici ces formes de construction de l'O/R name (les attributs entre crochets sont des attributs optionnels):

1)

Country Name
ADMD Name
[PRMD Name]
[Personal Name]
[Organization Name]
[Organization Unit Name]
[DDAs]
Au moins un des cinq attributs optionnels doit être sélectionné.

2)

Country Name
ADMD Name
Unique UA Identifier Name
[DDAs]

3)

Country Name
ADMD Name
X.121 Address
[DDAs]

4)

X.121 Address
[Telematic Terminal ID]

Remarque importante:

Pour les formes 1) et 2), tous les O/R names doivent comprendre un **ensemble d'attributs de base** (**base attribute set**, c.-à-d. un ensemble d'attributs permettant d'identifier un MD particulier). Cette obligation est due au fait qu'aucun service d'annuaire global (connaissant tous les attributs de tous les UAs du monde) n'était standardisé lors de l'élaboration de X400-84. Elle permet, sans l'utilisation de service d'annuaire, de déterminer directement vers quel MD doit être envoyé un message. Lorsque le message est arrivé dans le MD destinataire, un éventuel service d'annuaire local (propre au MD destinataire) donnera, si nécessaire, plus d'informations sur les attributs de l'UA destinataire (attributs standardisés, mais aussi DDAs). Voici l'ensemble des attributs de base reconnus par le MTS :

-si l'UA appartient à un ADMD:
Country Name + ADMD Name de l'UA.
-si l'UA appartient à un PRMD:
Country Name + ADMD Name associé au PRMD de l'UA.

I.1.4. Représentation en couches du service de messagerie X.400.

Le service de messagerie est une application du modèle OSI. Les fonctions de traitement de messages X.400 dans la couche application peuvent être divisées en deux sous-couches appelées UAL (UA Layer) et MTL (Message Transfert Layer). La sous-couche UAL comprend essentiellement les fonctions associées au contenu du message tandis que la sous-couche MTL comprend essentiellement les fonctions associées au transfert des messages.

Les entités et les protocoles liés au modèle en couches sont représentés à la figure 1.3.

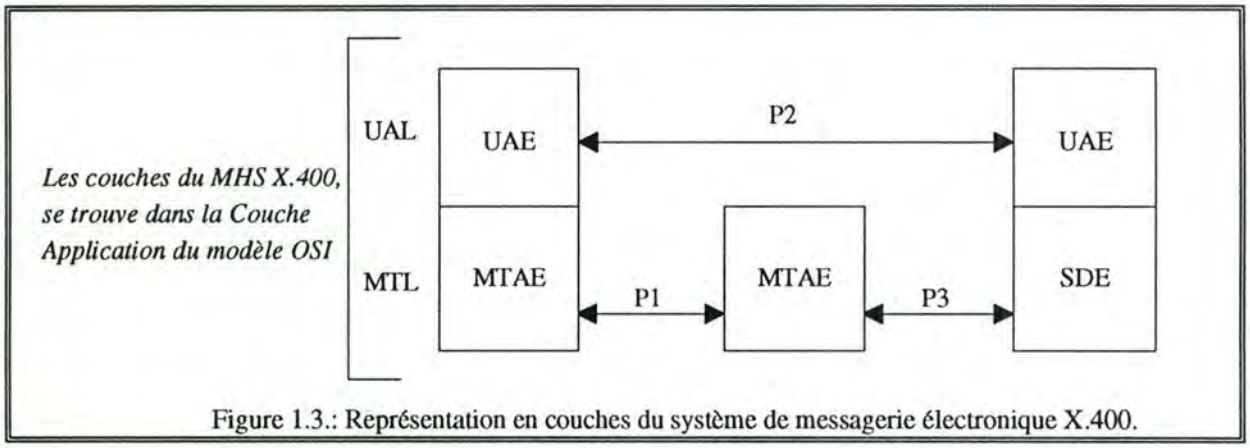

Figure 1.3.: Représentation en couches du système de messagerie électronique X.400.

L'entité UA (UAE) est l'entité fonctionnelle qui permet à l'UA de transférer des messages à d'autres UAs dans un UAL. Ce transfert se fait via un classe de protocoles (Pc), chacun d'eux définissant la syntaxe et la sémantique d'un type de message. Dans le cas de l'UA de la messagerie interpersonnelle, c'est le protocole P2 (décrit dans l'avis X.420) qui est utilisé entre les UAEs. (Interpersonal Messaging Protocol, IPM protocol).

L'entité MTA (MTAE) est l'entité fonctionnelle qui permet à l'UA de transférer des messages à d'autres UAs dans un UAL. Ce transfert se fait via le protocole P1 (décrit dans l'avis X.411). Lorsque l'UA et le MTA ne sont pas co-résidents, l'entité SDE (Submission and Delivery Entity) permet alors de servir d'intermédiaire entre l'UA isolé et le MTA éloigné (MTA public ou un MTA privé d'un VAN). Il permet à l'UA isolé de disposer des mêmes services (accès à distance) offerts par la couche MTL qu'un UA relié à un MTA co-résident. Pour cela le protocole P3 (décrit dans l'avis X.411) définit les interactions entre un SDE et un MTAE. La figure 1.3. montre la disposition de ces différents composants, ainsi que les protocoles utilisés dans l'IPMS.

I.1.5. Les services de transfert de messages (services offerts par le MTL).

Le MTL fournit des services de transfert de messsages (MT Services) à l'utilisateur du MTS afin de lui permettre d'accéder au MTS pour envoyer des messages d'un UA vers un autre UA ou vers plusieurs UAs, avec de nombreuses options appelées éléments de service de transfert de messages (MT Service Elements) décrits dans 4.1. de X.400 (X.400-84). Ces éléments de service permettent:

-la spécification des types d'informations encodées dans le message, avec des conversions éventuelles de types (les différents types d'informations encodées et les règles de conversion entre ces types sont décrites dans la norme X.408),
-la gestion d'accusés de livraison ou de non livraison de messages aux UAs destinataires,
-la possibilité de demander une livraison différée à telle heure/date,
-la protection de l'accès au MTA et à l'UA par mot de passe,
-la possibilité d'utiliser un message sonde (**probe message**) pour tester l'existence d'un UA,
-la possibilité pour un UA de spécifier au MTS quels types de message il peut reconnaître et se faire délivrer,
-l'identification des messages par un numéro,
-etc.

Une liste plus complète et plus précise de ces éléments de service, ainsi qu'une classification en cinq groupes (Basic MT service elements, Submission and delivery service elements, Conversion service elements, Query service elements et Status and inform service elements) sont décrites dans 4.1. de X.400: "Message Transfert Service".

I.1.6. Les services de la messagerie interpersonnelle (services offerts par l'IPM UAL).

Les services offerts par l'IPM UAL, appelés "IPM Services", permettent à un individu d'envoyer et de recevoir des IPMs. On distingue d'une part les services nécessitant la coopération et la coordination de plusieurs IPM UAs (ex: accusé de réception), et d'autre part les services ne nécessitant pas une telle coordination (traitement de texte pour composer les messages, facilités d'archivage et de consultation de messages). Les premiers doivent être standardisés et sont définis dans les recommandations X.400. Les seconds ne doivent pas être standardisés (ils sont appelés local IPM UA Services) et ne sont donc pas repris dans X.400. Ces services sont basés d'une part sur les éléments de service MT et d'autre part sur les éléments de service spécifiques à l'IPMS. L'ensemble de ces éléments de service forme les éléments de service de la messagerie interpersonnelle (IPM Service Elements). Voici quelques exemples d'IPM Service Elements standardisés et définis dans X.400:

-tous les services offerts par la couche MTL,
-possibilité d'attribuer un identifiant aux IPMs,
-possibilité d'inclure différents types de body part dans un IMP,
-des indications diverses sur l'IPM (sujet, date d'expiration, importance, sensibilité, référence à d'autres messages,...),
-possibilité de demander des accusés de **réception** ou de **non réception** à l'utilisateur destinataire (ou d'être avertis),
-déterminer si l'IPM reçu a été relayé automatiquement (auto-forwarded),
-etc.

Une liste plus complète et plus précise de ces éléments de service, ainsi qu'une classification en six groupes (Basic MT service elements, Submission and delivery, and conversion service elements, Cooperating IPM UA action service elements, Cooperating IPM UA information service elements, Query service elements et Status and inform service elements) sont décrites dans 4.2 de X.400 (X.400-84): "Interpersonnal Messaging Service".

I.1.7. Les messages échangés dans le MHS (entre UAs et entre MTAs).

Les UAEs s'échangent des unités de données appelées **UAPDUs** (**User Agent Protocol Data Units**, cf. X.420), en respectant un protocole de la classe **Pc** (**P2** dans le cas IPM UAs).

Les MTAEs s'échangent des unités de données appelées **MPDU** (**Message Protocol Data Units**, cfr X.411), en respectant le protocole **P1**. Lorsqu'un utilisateur désire envoyer un message à un autre utilisateur, son message est d'abord placé par son UA dans UAPDU. Cet UAPDU est alors passé au MTA (par la primitive Submit.Request) qui le considèrera comme un nouveau message à communiquer et l'enveloppera dans un MPDU.

I.1.7.1. Les UAPDUs.

Deux types d'UAPDUs sont définis au sein de l'IPMS: les **IP-messages** (**IM-UAPDUs**) et les **IPM-status-reports** (**SR-UAPDUs**).

I.1.7.1.a. Les IP-messages (IM-UAPDUs).

Un **IP-message** (ou **message P2**) est un UAPDU échangé entre IPM UAs et contenant l'information que l'utilisateur désire communiquer. L'IPM est divisé en deux parties: l'**en-tête** ("**heading**") et le **corps** ("**body**").

L'en-tête comporte des champs de données qui contiennent des informations relatives au corps. Voici une brève description des plus importants de ces champs:

-IPMessageIP: numéro identifiant de l'IP-message,
-Originator: l'O/R Name de celui qui a soumis l'IP-message,
-PrimaryRecipients: les O/R Names des destinataires principaux de l'IP-message,
-CopyRecipients (CC): les O/R Names des destinataires qui reçoivent une "copie-carbonne" de l'IP-message,
-BlindCopyRecipients (BCC): les O/R Names des destinataires secrets qui reçoivent une "copie-carbonne" de l'IP-message,
-InReplyTo: numéro identifiant de l'IP-message auquel cet IP-message répond,
-Obsoletes: les numéros identifiants des l'IP-messages qui sont rendus périmés par cet IP-message,
-Subject: le sujet de l'IP-message,
-ExpiryDate: la date d'expiration de l'IP-message,
-Importance: trois valeurs sont posssibles pour l'importance de l'IP-message: low, normal, high,
-Sensitivity: trois valeurs sont posssibles pour la sensibilité de l'IP-message: personal, private, company, confidential,
-Autoforwarded: champ indiquant si l'IP-message a été relayé automatiquement.

Le corps est le message proprement dit à communiquer. Le corps d'un IP-message est une séquence de body part. Chaque body part contient des données d'un seul type. La liste complète et précise des types de données possibles se trouve dans X.420, section 3.2.2. On y trouve par exemple: du texte, du fac-similé, de la voix, du télex, des dessins,...

I.1.7.1.b. Les IPM-status-reports (SR-UAPDUs).

L'**IPM-status-report** est un UAPDU créé par un IPM UA et envoyé à un autre IPM UA, soit pour être directement utilisé par lui-même, soit pour être directement communiqué à l'utilisateur de cet IPM UA (ex: notification de réception). Une description plus précise des IPM-status-report se trouve dans X.420, section 3.3.

I.1.7.2. Les MPDUs (X.411).

Deux types de MPDUs sont définis au sein du MTL: les **User-MPDUs** et les **Service-MPDUs**. Les User-MPDUs contiennent les messages à communiquer, alors que les Service-MPDUs, quant à eux, permettent de transférer des informations au sujet de ces messages (atteignabilité d'un utilisateur, livraison/non-livraison d'un message,...).

I.1.7.2.a. Les User-MPDUs.

Ce sont des MPDUs qui permettent le transfert des messages entre MTAs. Ils sont constitués de deux parties: d'une **enveloppe** ("**enveloppe**") et d'un **contenu** ("**content**").

Sur l'enveloppe figure les champs de données requises par le MTS pour l'envoi au destinataire. Voici une brève description des plus importants de ces champs:

-MPDUIdentifier: un numéro identifiant le MPDU (généré par l'UA pour la corrélation avec un éventuel rapport de livraison),
-Originator: l'O/R Name de l'expéditeur,
-RecipientInfo: l'O/R Name du(des) destinataire(s),
-ContentType: le type du contenu,
-OrigialEncodedInformationTypes: les types d'informations encodées (EITs).(Fac-similé,télex,...),
-Priority: la priorité du message.

Le contenu est un UAPDU passé par l'UA au MTA (IP-message ou IPM-status-report dans le cas du IPMS).

I.1.7.2.b. Les service-MPDUs.

Ce sont des MPDUs permettant de transférer des informations au sujet de l'atteignabilité d'un utilisateur (**Probe-Report-MPDUs**), de la livraison/non-livraison d'un message (**Delivery-report-MPDUs**). Les Probe-Report-MPDUs ne contiennent qu'une enveloppe et sont utilisés par la couche MTL pour fournir les éléments de service d'interrogation. Les Delivery-Report-MPDUs, quant à eux, sont générés par le dernier MTA

par où le message est passé et sont destinés à l'expéditeur du message. Ils sont constitués d'une enveloppe et d'une partie contenant les informations relatives à la livraison/non-livraison. Le contenu exact d'un Delivery-report-MPDU est donné dans X.411, section 3.4.3. (X400-84); celui d'un Probe-MPDU est donné dans X.411, section 3.4.4.(X.400-84).

I.1.7.3. Structure global d'un User-MPDU contenant un message interpersonnel (IM-UAPDU).

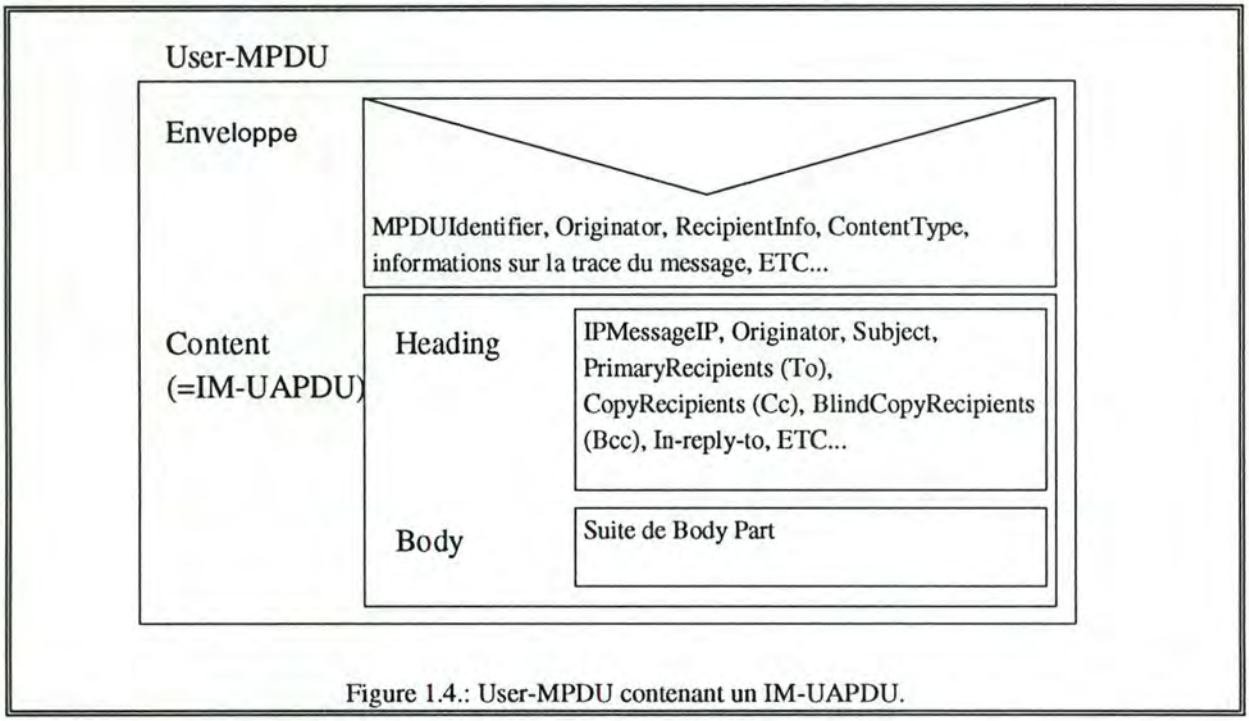

Figure 1.4.: User-MPDU contenant un IM-UAPDU.

I.1.8. Les éléments constituants du MTAE et du SDE.

La section I.1.4. décrivait la représentation en sous-couches du système de messagerie électronique X400: l'UAL et la MTL. Nous verrons dans cette section que chaque MTAE et chaque SDE se décomposent à leur tour en plusieurs sous-entités (fig. 1.5.).

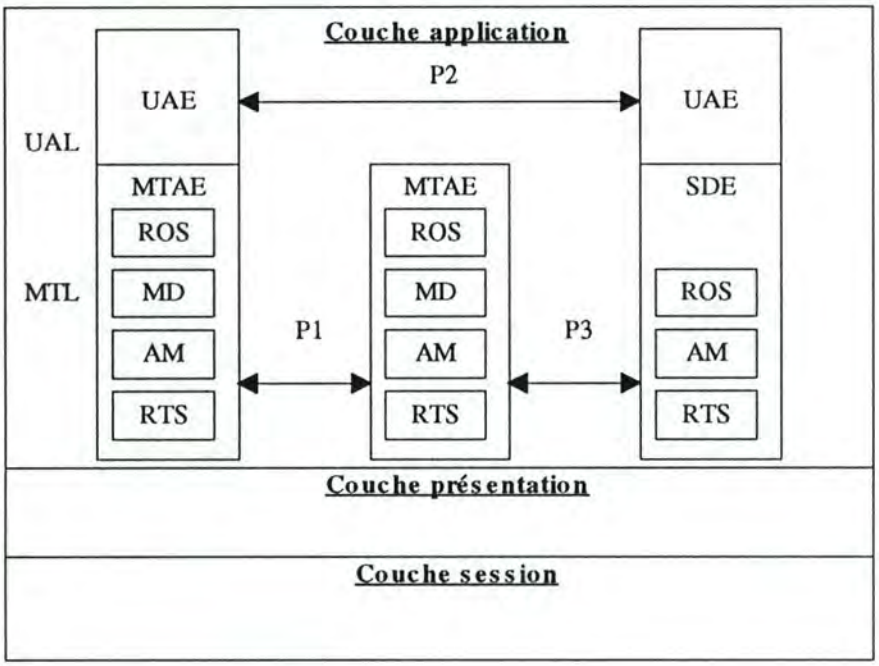

Figure 1.5.: Les éléments constituants les MTAEs et les SDEs.

I.1.8.1. Les éléments constitutifs du MTAE.

Une MTAE est composé de quatre éléments (fig. 1.5.): d'un **Message Dispatcher (MD)**, d'un **Association Manager (AM)**, d'un **Reliable Transfert Service (RTS)** et d'un **Remote Operation Service (ROS)**.

Le **Message Dispatcher (MD)** permet au MTA d'exécuter les actions prévues par le protocole P1 lorsque le MTA reçoit un MPDU d'un autre MTA ou lorsqu'un de ses UAs lui soumet un message. Les actions de transfert proprement dit de messages de et vers d'autres MTAs n'en font pas partie .

L'**Association Manager (AM)** permet au MTA d'établir, de contrôler et de fermer les associations fournies par le RTS. C'est l'Association Manager qui décide quand et quelle(s) association(s) ouvrir et fermer, assigne une priorité aux MPDUs à transférer, décide quelle(s) association(s) utiliser pour le transfert de MPDUs (en fonction de leur priorité),...

Le **Reliable Transfert Service (RTS)** permet au MTA d'obtenir des associations pour le transfert fiable de MPDUs avec d'autres MTAs. Le RTS constitue la couche inférieure de la couche 7. Il utilise les services offerts par la couche 5 (session), qui lui sont fournis via la couche 6 (présentation) (X.410, section 4 de X.400-84). La partie de la couche 7 utilisant le RTS est appelé le **RTS-user**.

Le **Remote Operation Service** (**ROS**) permet au MTAE et au UAE isolé (par l'intermédiaire du SDE) de communiquer entre eux. Le protocole permettant la communication entre MTAE et SDE est, comme nous l'avons déjà vu, le protocole P3. Il permet à chacune de ces deux entités d'exécuter des opérations à distance sur l'autre entité (**remote operations**). L'entité "client" envoie un message à l'entité "serveur" contenant la description de l'opération à exécuter et ses paramètres. L'entité "serveur" exécute alors l'opération et renvoye à l'entité "client" un message contenant les résultats de l'opération (message résultat ou message d'erreur). Les unités de données échangées entre MTAE et SDE sont appelées **OPDUs** (**Operation Protocol Data Units**). Elles sont classées en **Invoke OPDU** lorsque le "client" requiert l'exécution d'une opération, en **ReturnResult OPDU** lorsque l'opération s'est correctement déroulée et que les résultats de l'opération sont retournés au "client" de l'opération et en **ReturnError OPDU** lorsque le "serveur" signale au "client" de l'opération a échouée.

I.1.8.2. Les éléments constitutifs du SDE.

Une SDE est composée de trois éléments (fig. 1.5.): d'un **Remote Operation Service** (**ROS**), d'un **Association Manager** (**AM**) et d'un **Reliable Transfert Service** (**RTS**). Les opérations effectuées par le MD ne sont pas effectuées dans la SDE.

I.1.9. Les primitives de service.

Voici un résumé des primitives de service offertes à l'UAE par le MTAE et par le SDE. Les primitives de service sont reroupées par rapport à leur fonctionnalité, en indiquant la chronologie des événements: **Req** = Request, **Ind** = Indication, **Resp** = Response, **Conf** = Confirm.

I.1.9.1. Les primitives de service offertes à l'UAE par le MTAE.

Le tableau 1.2. reprend les primitives de service offertes à l'UAE par le MTAE. Une desription plus complète se trouve dans X.411, section 2.2. de X.400-84.

Nom	Initiateur	Evénements	Description du service
LOGON	UA	Req/Conf	l'UA veut initialiser une interaction avec le MTA
LOGON	MTA	Ind/Resp	le MTA veut initialiser une interaction avec l'UA
LOGOFF	UA	Req/Conf	l'UA veut terminer l'interaction avec le MTA
REGISTER	UA	Req/Conf	l'UA veut modifier certains de ses paramètres utilisés par le MTA
CONTROL	UA	Req/Conf	l'UA veut modifier les restrictions concernant le contrôle des messages que la MTS peut lui envoyer
CONTROL	MTA	Ind/Resp	le MTA indique à l'UA quels messages de l'UA seront acceptés par la MTL
SUBMIT	UA	Req/Conf	l'UA soumet un message au MTS
PROBE	UA	Req/Conf	l'UA désire envoyer une sonde (probe message)
DELIVER	MTA	Ind	le MTA livre un message à l'UA
NOTIFY	MTA	Ind	le MTA livre un accusé de livraison/non-livraison à l'UA
CANCEL	UA	Req/Conf	l'UA demande l'annulation de la livraison d'un message précédemment posté avec l'option "Deferred Delivery" (livraison après telle date/heure)
CHANGE-PASSWORD	UA	Req/Conf	l'UA indique au MTA le nouveau mot de passe de l'UA
CHANGE-PASSWORD	MTA	Ind	le MTA indique à l'UA le nouveau mot de passe du MTA

Tableau 1.2.: Primitives de service offertes à l'UAE par le MTAE.

I.1.9.2. Les primitives de service offertes au RTS-user par le RTS.

Le tableau 1.3. reprend les primitives de service offertes au RTS-user par le RTS. Une description plus complète se trouve dans X.410, section 3 de X.400-84.

Nom	Initiateur	Evénements	Description du service
OPEN	RTS-user	Req/Ind/Resp/Conf	Le RTS-user veut établir une nouvelle association avec un autre RTS-user
CLOSE	RTS-user	Req/Ind/RespConf	Le RTS-user ayant l'initiative veut fermer une association
TURN-PLEASE	RTS-user	Req/Ind	Un RTS-user n'ayant pas l'initiative demande pour l'avoir
TURN-GIVE	RTS-user	Req/Ind	Un RTS-user ayant l'initiative la cède à l'autre RTS-user
TRANSFER	RTS-user	Req/Ind	Le RTS-user demande le transfert d'un APDU
EXCEPTION	RTS	Ind	Le RTS signale au RTS-user que le transfert d'un APDU n'a pu se faire comme demandé

Tableau 1.3.: Primitives de service offertes au RTS-user par le RTS.

I.1.9.3. Les primitives de service offertes à l'UAE par la SDE.

Rappelons qu'une SDE doit permettre à une UAE d'avoir accès au MTS comme si l'UAE était reliée directement à la MTAE. Par conséquent, une SDE doit offrir les mêmes

services qu'une MTAE.

Rappelons également que chacun des services offerts à l'UAE par la SDE est associé à une opération distante effectuée par la MTAE ou la SDE.

Le tableau 1.4. reprend les opérations (les OPDUs) et les primitives de service offertes à l'UAE par le SDE.

Nom	Initiateur	Evénements	Opérations
REGISTER	UA	Req/Conf	Register
CONTROL	UA	Req/Conf	Control
CONTROL	MTA	Ind/Resp	Control
SUBMIT	UA	Req/Conf	Submit
PROBE	UA	Req/Conf	Probe
DELIVER	MTA	Ind	Deliver
NOTIFY	MTA	Ind	Notify
CANCEL	UA	Req/Conf	Cancel
CHANGE-PASSWORD	UA	Req/Conf	Change-password
CHANGE-PASSWORD	MTA	Ind	Change-password

Tableau 1.4.: Primitives de service et opérations offertes à l'UAE par le SDE.

Remarque:

Aucune opération ne correspond aux primitives de service LOGON et LOGOFF: l'UAE ne pouvant interagir avec une MTAE qu'au moyen d'opérations à distance, ne pouvant ainsi ouvrir qu'une association au niveau RTS.

I.2. Description des nouveaux concepts introduits par X.400-88.

a)Le message Store et les Access Units.

Deux nouvelles entités fonctionnelles sont introduites dans X.400-88: les messages Store ("réserve de messages") et les Access Units ("unités d'accès").

-le message Store (MS) est défini par la nouvelle recommandation X.413 et permet à un UA isolé d'accéder à son MTA avec beaucoup plus de contrôle sur les interactions UA / MTA que ne le permet le protocole P3. Généralement, le MS sera situé sur le même système que le MTA. Le protocole entre le MS et l'UA est appelé protocole P7. Auparavant, l'UA isolé était forcé d'accepter les messages lorsqu'il ouvrait une connexion avec le MTA en utilisant le protocole P3. Aujourd'hui, l'UA peut utiliser le protocole P7 pour contrôler lui-même la livraison des messages qui lui sont destinés. Il peut également demander que ses messages soient placés temporairement dans son MS. Le MS est une véritable base de données des messages et des rapports de livraison/non-livraison.

-les Access Units sont de deux types: les Access Units pour les services télématiques (Access Units for Telematic Service) et les Access Units pour la livraison physique (Physical Delivery Access Units).

b)Les mécanismes d'extention.

X.400-88 a introduit des mécanismes permettant de définir, à volonté, des extentions aux protocoles de base. Celles relatives à la sécurité des systèmes de messagerie électronique nous intéressent plus particulièrement et seront abordées au chapitre V.

c)La naissance d'un nouveau concept: les services abstraits.

La version 1988 précise les services offerts par les différentes sous-couches du niveau 7 grâce au concept de service abstrait. Un double objectif était visé avec ce concept. Tout d'abord rendre possible la description d'un service complet par la méthode orientée objet, et ensuite d'affiner ce service jusqu'au niveau de detail exigé pour décrire correctemnt le système. Une caractéristique puissante de l'approche service abstrait est qu'elle permet à la fois une description graphique et une description sous forme de notation ASM.1. Cette dernière forme autorise la réutilisation des paramètres de service dans les spécifications du protocole et par conséquent réduit d'une manière significative la duplication d'information de les textes des services et protocoles correspondant.

La version graphique des services abstraits ainsi que leur description en notation ASN.1 sont documentées dans la recommandation X.407. La représentation graphique décrit les objets par une méthode décende communiquant avec, soit un environnement complet, soit un système complet comme objet le plus important. Cela peut se décrire par un ovale contenant d'autres ovales ou boîtes représentant les objets du niveau inférieur immédiat. Un ovale plus petit a l'intérieur d'un ovale indique que l'on a à faire à un sous-sytème appartenant au système principal. C'est de cette manière qu'est représenté le système de transfert de messages (MTS) relativement au système de messagerie personne à personne (IPMS). L'ancien modèle en sous-couches a donc été remplacé par un modèle en "oignon" avec la même sémantique. Les lignes reliant des symboles d'objets illustrant les connexions entre ce que l'on appele point d'accès. Un objet peut donc avoir un ou plusieurs points chacun offrant un ensemble de services abstraits particulier. Un point d'accès peut, soit fournir des services à partir d'un objet, soit utiliser les services d'un autre objet. Les services proposés par un point d'accès sont exécutés sous la forme d'opérations qui peuvent être invoquées à partir d'autres objets.

Chapitre II:

ALBERT, un langage de spécification des besoins orienté agent.

II.1. Introduction générale.

Pour la phase de **D**esign **E**ngineering (DE), la méthode transformationnelle est une approche récente préconisée par les spécialistes du génie logiciel. Cette méthode repose sur la transformation de descriptions formelles (c.-à-d. ayant une sémantique mathématique et/ou logique) pour générer de manière assistée les différentes transformations jusqu'au code source. Le Design Engineering comporte trois phases:

1°)la phase de **spécification** du logiciel: a partir des besoins des utilisateurs (généralement définis de manière informelle dans le cahier des charges), on élabore une spécification formelle qui correspond à une définition précise et consistante des attentes des utilisateurs vis-à-vis du logiciel,

2°)la phase de **conception (design)**: on exprime une solution abstraite au problème décrit par la phase de spécification. Cette solution doit être correcte, indépendante des choix particuliers de réalisation et exprimée formellement. La conception se décompose en deux autres phases:

- la **conception globale**, qui se penche sur l'architecture des modules,
- la **conception détaillée** du fonctionnement de chaque module,

3°)la phase d'**implémentation**: permet d'élaborer le code source de l'application.

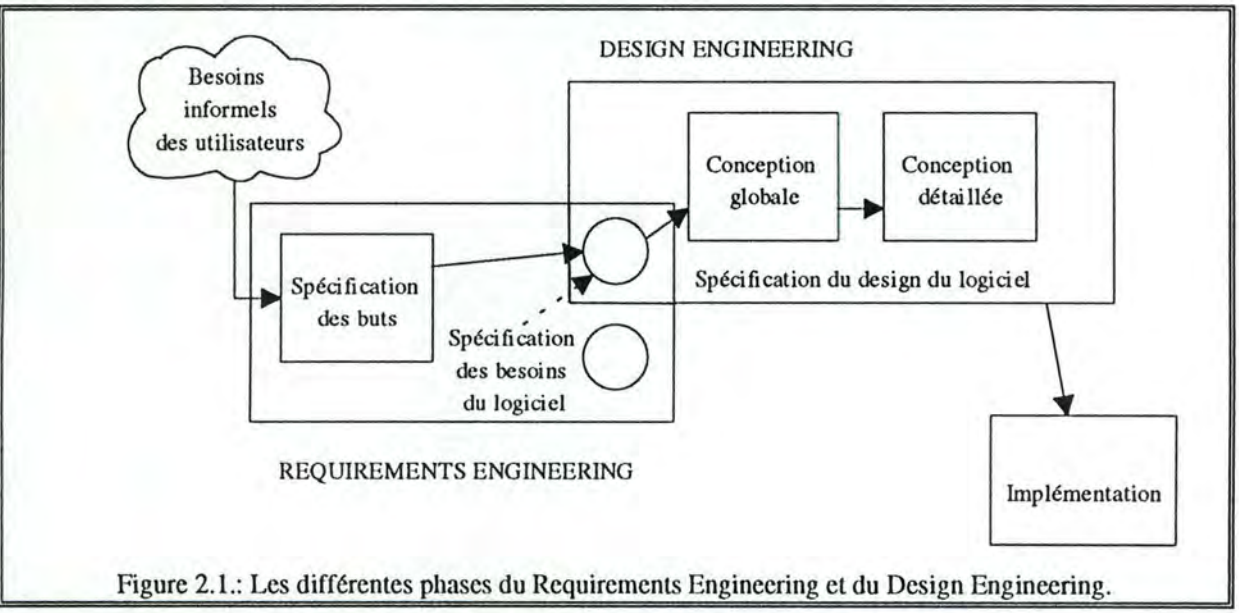

Figure 2.1.: Les différentes phases du Requirements Engineering et du Design Engineering.

Cependant, pour un système complexe (système composite) incluant des procédures manuelles, du hardware, des "devices" spécifiques et des éléments logiciels où tous ces éléments interagissent, il s'est avéré essentiel d'étendre la première phase de spécification et d'inclure en amont du Desing Engineering une nouvelle discipline, appelée **R**equirements **E**ngineering (RE). Celle-ci permet d'élaborer, à partir des souhaits informels des utilisateurs, un document contenant les spécifications des besoins. Ce document ne comporte pas uniquement les spécifications des parties logicielles, mais également les spécifications de l'environnement ("devices", hardware, utilisateurs humains,...) ainsi que les spécifications des interactions entre ces deux-ci. Dans l'approche proposée dans [DDPM93], la phase d'analyse des besoins comporte deux phases:

1°)la phase de **spécification des buts** que doit atteindre le système à développer: par exemple, dans un système de messagerie électronique on peut dire que la phrase "quand un message est envoyé, il sera délivré dans le futur" est un but du système à mettre en oeuvre,

2°)la phase d'élaboration du document des **spécifications des besoins** : on décrit les différents composants du système ainsi que leurs "comportements" pour atteindre les buts du système à développer. Par exemple, le cahier des charges du système de messagerie électronique identifiera les différents composants (UA, MTA, DS, User,...) et décrira la manière dont ils interagiront entre eux pour atteindre le but fixé.

Remarquons que pour chaque partie logicielle de ce document, une phase de Design Engineering pourra être appliquée en utilisant comme spécification la spécification des besoins du logiciel. Pour permettre de pouvoir encore utiliser la méthode transformationnelle, cette phase devra être décrite à l'aide d'un langage formel de spécification.

Remarquons également que si l'on désire, dans le cycle de vie d'un système composite, modifier un élément, que celui-ci soit ou ne soit pas logiciel, il faudra retourner soit à la spécification du logiciel, soit à la spécification des buts. Il faudra en outre réexécuter l'approche transformationnelle à partir du point de retour. Dans le cas où la modification ne fait pas intervenir un élément de l'environnement, ce sera à la spécification du logiciel qu'on retournera. En effet, seul un élément du logiciel pourra être modifié, et par ailleurs, cette modification ne pourra en aucun cas avoir d'incidence sur le comportement de l'environnement. Ce sera, par contre, à la spécification des buts qu'on retournera, dans le cas où la modification du système porte sur l'environnement ou si celle-ci, portant sur le logiciel, agit directement sur le comportement de l'environnement. Par exemple si l'on désire modifier un élément du protocole entre l'UA et le MTS n'entraînant pas une modification du comportement de l'utilisateur, ce sera à la phase de spécification du logiciel qu'on remontera pour décrire cette modification. Par contre, si l'on désire ajouter des services de sécurité à un système existant, il est sûr que le comportement de l'environnement n'aura pas été spécifié en tenant compte d'éventuelles attaques pouvant survenir sur le système. De plus, il faudra également décrire les buts du système où seront par conséquent modifiés les comportements des utilisateurs permettant de se protéger face à ces éventuelles attaques. Il faudra, dans ce cas, revenir à la spécification des buts. Cela justifie la description et l'utilisation du langage ALBERT décrit au paragraphe suivant.

II.2. Introduction au langage de spécification des besoins ALBERT.

Le langage présenté dans ce chapitre et utilisé par la suite (chapitres III et VI) est un **langage formel de spécification des besoins** basé sur la logique du premier ordre typée avec des extensions **déontique, épistémique** et **temporelle**: il est nommé ALBERT (**A**gent-oriented **L**anguage for **B**uilding and **E**liciting **R**equirements for real-**T**ime systems). Ce langage a été développé par l'équipe du Professeur Dubois, à l'Institut d'Informatique des Facultés Universitaires Notre-Dame de la Paix (Namur, Belgique). Le langage **ALBERT** permet de définir les **comportements admissibles** d'un système **composite** ([DDDPM93], [DDP93]).

II.2.1. Caractéristiques générales du langage ALBERT.

Toute spécification ALBERT se structure en terme d'agents. Le concept "agent" peut être vu comme une spécialisation du concept "objet", caractérisé par des propriétés de responsabilité et de perception. Chaque agent est décrit au moyen des concepts suivants:

1°)le concept d'**état** (state) interne: l'état permet de représenter la connaissance que possède l'agent du monde extérieur,

2°)le concept d'**actions**: les actions servent à modéliser les changements d'états et les événements sans influence directe sur l'état de l'agent,

3°)Le concept de **perception:** la perception permet à un agent d'avoir accès, en partie ou en totalité, à l'état d'un autre agent et de percevoir des actions exécutées par un autre agent,

4°)le concept de **contraintes**: les contraintes permettent à l'analyste de définir les **comportements admissibles** du système qu'il décrit. On peut lier à un agent un **ensemble de vies possibles**, modélisant ainsi tous les comportements admissibles pour cet agent. Une **vie** est une succession (une séquence finie ou infinie) alternée de changements et d'états, chaque état étant désigné par une valeur temporelle qui ne cesse d'augmenter tout au long de la vie. Enfin, pour en finir avec les définitions concernant les comportements admissibles, notons qu'un changement est composé de plusieurs occurrences d'actions simultanées et qu'une histoire est la séquence des changements se déroulant dans une vie possible de l'agent. La figure 2.2. donne un exemple de vie possible pour un MTA du système de messagerie électronique X.400. Pour fournir à l'analyste une méthode, les contraintes sont regroupées en trois familles:

-contraintes de **base** (composantes dont la valeur dérive d'une autre composante et conditions initiales),
-contraintes **locales** (contraintes sur l'évolution de l'état, sur les effets des actions, sur la causalité entre actions, sur la capacité d'exécuter telles ou telles actions dans certaines conditions),
-contraintes de **coopération** entre agents (contraintes sur la perception des états et des actions, sur l'information montrée à l'extérieur).

Nous détaillerons davantage les contraintes dans le paragraphe II.2.2. consacré à la "phase de spécification des contraintes".

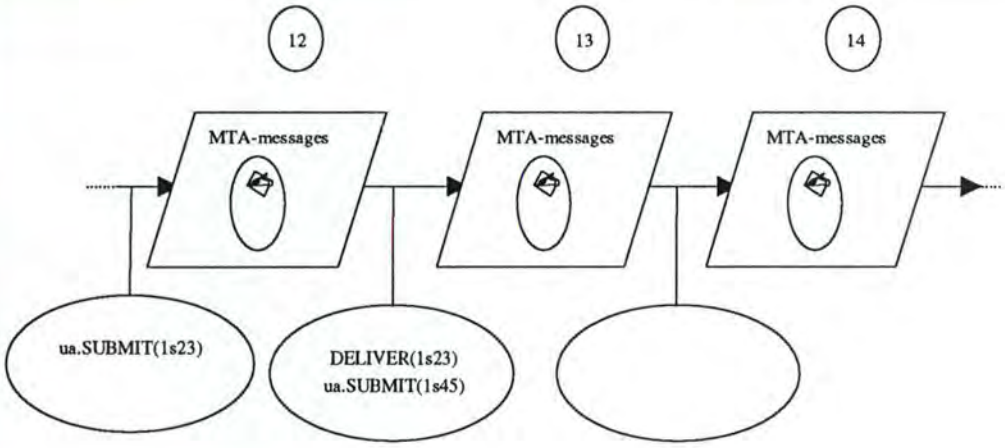

Figure 2.2.: Une vie possible pour un agent MTA.

L'agent est l'unité de structuration des spécifications. Au cours du processus de spécification, un agent dont le comportement a été décrit peut se subdiviser en sous-agents dont la combinaison des comportements correspond à celui de l'agent décomposé. Une spécification achevée apparaîtra donc comme une hiérarchie d'agents. Une spécification se compose de deux parties:

-la partie contenant les **déclarations** qui permettent d'introduire le vocabulaire du système étudié (composantes d'états, actions,...). Cette partie de la spécification est décrite au moyen d'un formalisme **graphique**,

-la partie contenant les **contraintes** qui permettent de définir les comportements admissibles du système étudié au moyen d'expressions logiques. Cette partie de la spécification est décrite de manière **textuelle**.

II.2.2. Déclarations.

II.2.2.1. Déclaration de la hiérarchie des agents.

La partie "déclarations" de la spécification commence par l'identification des agents impliqués dans le système à réaliser. Les agents sont regroupés en sociétés élémentaires. Les sociétés peuvent à leur tour être regroupées en sociétés plus importantes. La hiérarchie dans laquelle ces agents s'inscriront résulte de cette enchevêtrement de sociétés. La figure 2.4. illustre la hiérarchie des agents dans le cas d'une messagerie électronique de type X.400. Remarquons par ailleurs, qu'une description en termes de société n'est pas dénuée de sens dans le cas des composantes de X.400. En effet, la version graphique des services abstraits définis dans X.400-88 utilise un même type de concept.

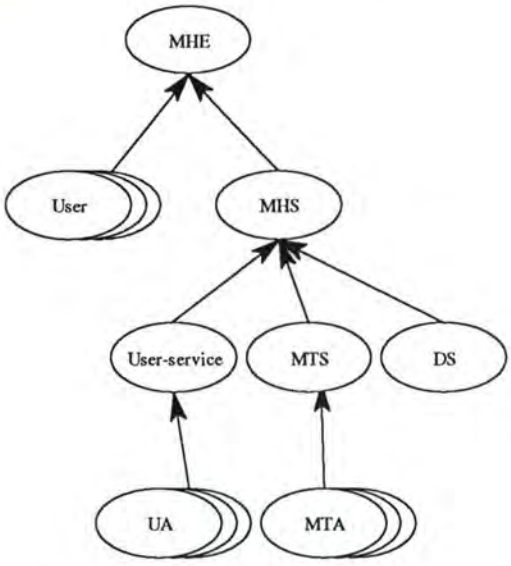

Figure 2.3: Représentation de la hiérarchie des agents dans le cas d'une messagerie électronique de type X.400.

On distingue deux types d'agents: les agents **simples** (ou **indécomposables**) et les agents **complexes**:

-les agents **simples** correspondent aux feuilles de l'arbre. Ces agents sont des composants terminaux pour lesquels l'analyste doit spécifier le comportement,

-les agents **complexes** correspondent à des noeuds non terminaux de la hiérarchie. Ils sont composés d'autres agents plus simples. Ces agents sont des composants virtuels parce qu'ils n'ont pas d'existence en soi et qu'ils sont juste des agrégats d'agents plus simples. Leur comportement se déduit du comportement des sous-agents. Les noeuds non terminaux sont construits au moyen de deux constructeurs de type: l'ensemble et le produit cartésien. L'ensemble (Set) permet de définir une classe d'agents à partir d'un agent simple, alors que le produit cartésien (Cartesian Product) permet de construire de nouveaux agents à partir d'agrégats d'agents existants.

La figure 2.3. illustre ce que nous venons de dire: les agents MHE, MTS et US, sont des agents complexes. L'ensemble des agents User, UA et MTA forment des classes.

II.2.2.2. Déclaration de la structure de l'agent.

La structure d'un agent consiste en la description d'une **structure d'état** et d'un ensemble d'**actions** pouvant se produire pendant la vie de l'agent. A la figure 2.4., les informations qui se trouvent à l'intérieur du parallélogramme sont sous la responsabilité de l'agent décrit. Celles situées à l'extérieur du parallélogramme proviennent d'autres agents qui décident de les exporter vers cet agent. Le nom de l'agent est mentionné au dessus du parallélogramme.

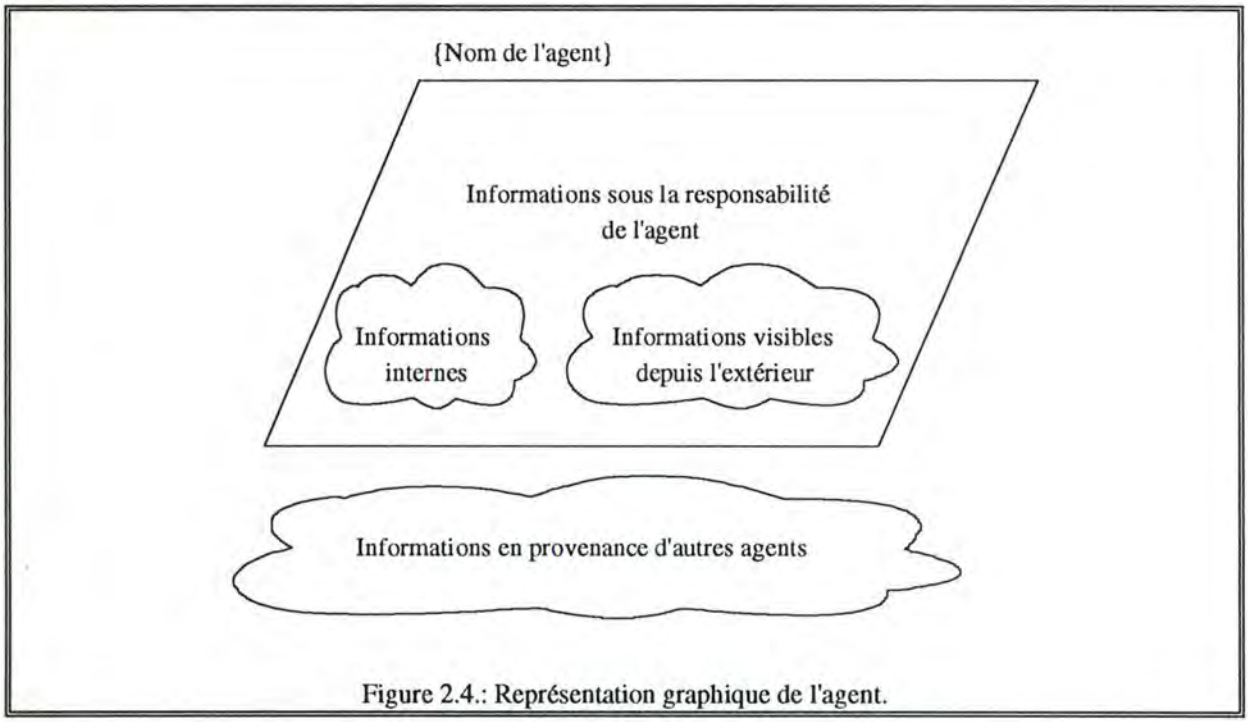

Figure 2.4.: Représentation graphique de l'agent.

II.2.2.2.1. Déclaration de la structure d'état.

L'état est en fait constitué de deux types de composantes d'état:

1°)les **instances**,
2°)les **populations**: ces populations sont en générale des ensembles d'instances, mais elles peuvent également être organisées sous forme de "suites" (sequences) ou de "tableaux" (tables).

Les éléments des composantes (c.-à-d. les instances) sont typés. Les types utilisés sont les suivants:

-les types de données **élémentaires prédéfinis** (STRING, BOOLEAN, INTEGER,...) avec leurs opérations usuelles,

-les **types élémentaires** définis par l'analyste, pour lesquels aucune structure n'est donnée. Une opération leur est accessible: le prédicat d'égalité,

-les types plus **complexes** construits par l'analyste (MESSAGE, REPORT, MESS-TOKEN,...) en utilisant un ensemble de constructeurs de types prédéfinis comme l'ensemble (set), la suite (list), le produit cartésien (cartesian product), et un ensemble de types élémentaires. En plus des opérations héritées des types intermédiaires, de nouvelles opérations peuvent être définies sur ces nouveaux types (cf. Annexe 1). Un constructeur, appelé "extension" (notée*), permet d'ajouter une valeur spéciale "UNDEF" à un type de données. Par exemple, une variable de type MESSAGE* peut prendre n'importe quelle valeur de type MESSAGE ou une valeur spéciale "UNDEF",

-les types décrivant les **identifiants** des agents. Les agents possèdent un mécanisme de clés qui permet d'identifier n'importe quel agent. Un type est automatiquement associé à chaque classe d'agents. Par exemple, à la figure 2.3., chaque agent User possède un identifiant de type USER. A l'intérieur de la description d'un agent une constante, appelée self, permet de se référer à son propre identifiant.

Chaque composante d'état possède les propriétés suivantes:

-elle peut être **constante** ou **variable** (time varying, c.-à-d. que cette composante d'état),
-elle peut être dérivée d'une autre composante d'état (**composante dérivée**), la valeur de cette dernière dépendant de la composante initiale.

La figure 2.5. donne la représentation graphique des différents types de composantes d'état ainsi que les propriétés que celles-ci peuvent prendre.

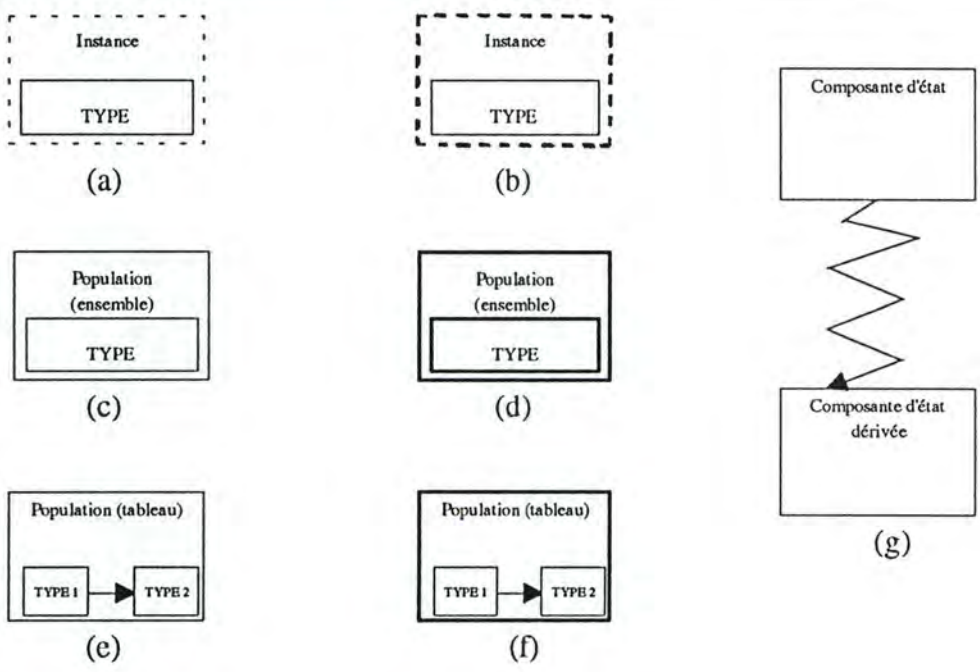

Figure 2.5.: Représentation graphique: (a) d'une instance variable; (b) d'une instance constante; (c) d'une population variable formée avec le constructeur SET; (d) d'une population constante formée avec le constructeur SET; (e) d'une population variable formée avec le constructeur TAB; (f) d'une population constante formée avec le constructeur SET; (e) d'une population constante formée avec le constructeur TAB et d'une composante d'état dérivée.

Ces structures d'état peuvent être exportées et importées.

Le langage ALBERT permet également de définir des attributs spécifiques en plus de la structure de type définie pour chaque instance (isolée ou inclue dans une population). Ces attributs peuvent se rapporter à chaque instance ou à une population bien précise.

Illustrons cette distinction par un exemple: à la figure 2.7., nous avons défini un type de données **PERSONNE**, dont la structure se compose de **Nom** et **Prénom**, caractéristiques communes à toutes les personnes. Nous avons défini également une population **étudiants**, (un set d'étudiants) dont le type de chaque instance est **PERSONNE**. Définissons encore deux

attributs: un attribut propre aux personnes étudiantes (numéro de carte) et se rapportant donc à chaque **instance** de la population **étudiants** et un attribut de la population **étudiants**, qui est unique pour toute la population (titulaire).

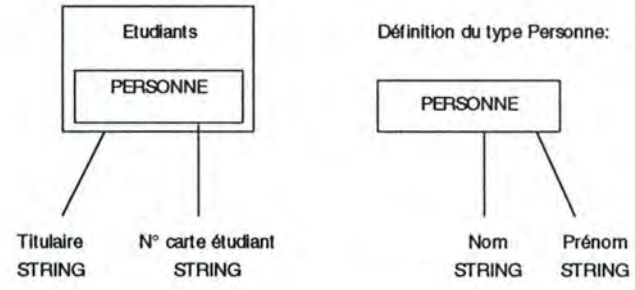

Figure 2.7.: Les attributs de la classe "PERSONNE" et l'attribut propre aux étudiants.

II.2.2.2.2. Déclaration d'une action.

Les actions peuvent avoir des arguments, ce qui permet de transférer de l'information d'un agent vers un autre. Les types des arguments sont similaires à ceux utilisés pour la description des composantes d'état. Pour différencier une action visible depuis l'extérieur d'une action purement interne, on utilise des représentations graphiques différentes (fig. 2.8.(a), (b), (c)):

-la figure 2.8.(a) permet de représenter une action purement interne à un agent,

-la figure 2.8.(b) permet de représenter une action se situant dans le parallélogramme de l'agent et trouve ainsi son origine au sein de celui-ci. Cette action sera dénommée par la suite "action interne". De celle-ci partira une flèche vers une zone contenant l'ensemble des agents pouvant percevoir cette action,

-la figure 2.8.(c), représente une action se situant en dehors du parallélogramme de l'agent et trouve ainsi son origine au sein d'un autre agent. Cette action sera dénommée par la suite "action externe". A celle-ci aboutira une flèche provenant d'une zone contenant l'agent d'où elle est originaire.

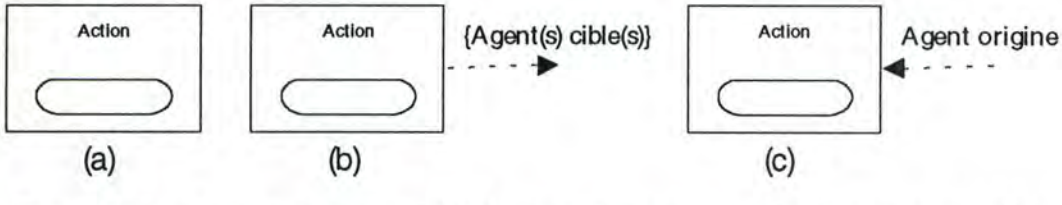

Figure 2.8: Représentation graphique: (a) d'une action non perceptible par un autre agent; (b) d'une action exportée vers d'autres agents; (c) d'une action importée.

II.2.3. Spécification des contraintes.

Le mécanisme permettant de réduire l'ensemble infini des vies possibles à une collection finie plus proche de l'évolution du système à réaliser est la **spécification des contraintes**. En langage ALBERT, les contraintes sont des énoncés en logique temporelle, épistémique et déontique qui identifient les comportements acceptables du système et qui en excluent les autres.

Nous allons à présent exposer les différentes familles de contraintes que l'analyste doit utiliser pour décrire le plus correctement possible le fonctionnement d'un système avec le langage ALBERT. Ces familles sont au nombre de trois:

-les **contraintes de base (Basic Constraints)**,
-les **contraintes locales (Local Constraints)**,
-les **contraintes de coopération (Cooperation Constraints)**.

Au sein de chaque famille, une classification en sous-catégories est opérée. Chaque sous-catégorie porte un nom et décrit un aspect bien particulier de la famille de contraintes à laquelle elle appartient. Ces sous-catégories sont au total au nombre de dix. Nous les exposerons au fur et à mesure de la description des différentes familles de contraintes. Décidons d'utiliser dès à présent la dénomination anglaise tant pour la description des différentes familles que pour celle des sous-catégories.

II.2.3.1. Les contraintes de base (Basic Constraints).

Les contraintes de base sont utilisées pour décrire l'état initial d'un agent (sous-catégorie **Initial Components**) ou pour donner des règles de dérivations pour les composantes dérivées (sous-catégorie **Derived Components**).

II.2.3.2. Les contraintes locales (Local Constraints).

Les contraintes locales sont utilisées pour décrire le comportement interne de l'agent. Quatre sous-catégories forment cette famille de contraintes: **State Behaviour**, **Effects of Action**, **Causality** et **Capability**.

a) **State Behaviour.**

Les contraintes de cette sous-catégorie permettent d'exprimer des propriétés sur les états de l'agent. Ces propriétés peuvent être d'ordre temporel (permettant de relier les états d'une vie admissible) ou purement statique (permettant d'exprimer des invariants). Pour exprimer les propriétés statiques, les règles usuelles de la logique du premier ordre fortement typé sont utilisées. Ce sont des expressions formées à partir des connecteurs logiques $\neg$ (not), $\wedge$ (and), $\vee$ (ou), $\Rightarrow$ implique, $\Leftrightarrow$ (ssi), $\forall$ (pour tout), $\exists$ (il existe). Par contre, pour exprimer les propriétés d'ordre temporel sur l'évolution du système, il nous faut utiliser des connecteurs temporels. Syntaxiquement, ceux-ci précèdent l'expression pour laquelle on pourra suivre l'évolution de son interprétation.

La table suivante décrit les différents connecteurs temporels utilisés dans le langage ALBERT.

$\Diamond \phi$	ϕ peut se révéler être vrai dans le présent ou le futur
$\blacklozenge \phi$	ϕ a pu s'être révélé vrai dans le présent ou le passé
$\Box \phi$	ϕ est vrai et le restera dans le futur
$\blacksquare \phi$	ϕ est vrai et le fut dans le passé
ϕ U ψ	ϕ est vrai et le restera jusqu'au moment où ψ le sera
ϕ S ψ	ϕ est vrai depuis que ψ le fut

Tableau 2.1: Les connecteurs temporels du langage ALBERT

Certaines contraintes permettent de décrire des propriétés relatives au temps réel. Il est donc nécessaire de décrire les délais et les "time-outs", comme par exemple l'expression suivante: un message ne pourra être contenu plus de trois jours dans un même MTA. Cela devient possible en apposant en indice une période temporelle sur les connecteurs temporels. Cette période temporelle possède une métrique choisi parmi les suivantes: Sec, Min, Hours, Days, ...

b) **Effects of Actions.**

Dans cette sous-catégorie seuls sont décrits les effets qui occasionnent un changement d'état dans la vie de l'agent. Lorsqu'une action est exécutée par un agent, les effets de celle-ci, si effets il y a, concernent la modification d'une ou plusieurs composantes d'état qui sont sous la responsabilité de cet agent.

c) **Causality.**

Les contraintes de cette sous-catégorie permettent d'exprimer les relations de causalité existant entre plusieurs occurrences d'action.

Exprimer des règles de causalité à l'aide des connecteurs habituels de la logique temporelle peut s'avérer assez complexe. Le langage ALBERT, grâce au connecteur $\xrightarrow{\Diamond}$ qui lui est spécifique, permet d'exprimer aisément ces critères de causalité. Par exemple, grâce à ce connecteur, il sera plus facile d'exprimer la contrainte " l'action act1 doit être exécutée par l'agent si celui-ci perçoit l'occurrence d'une autre action act2". Cette contrainte s'écrit: ag.act2 $\xrightarrow{\Diamond}$ act1. Au symbole $\xrightarrow{\Diamond}$, on peut adjoindre une quantification temporelle permettant d'exprimer des contraintes de performance. Par exemple ag.act2 $\xrightarrow{\Diamond \leq 10'}$ act1 signifie que l'action act1 devra être réalisée au plus tard 10 minutes après la perception de l'action act2. Dans le langage ALBERT, ce qui se trouve à droite du connecteur de causalité ne peut contenir que des actions sous la responsabilité de l'agent. Il peut exister des deux côtés du connecteur autant d'occurrences d'action que l'on désire. Dans le cas ou il en existe plusieurs, les occurrences doivent être regroupées de la façon suivante:

-act1;act2: signifie que l'occurrence de act1 est suivie de celle de act2,
-act1 $\otimes$ act2: signifie que l'occurrence de act1 a lieu au même moment que celle de act2,
-act1 $\|$ act2: signifie que les occurrences de act1 et de act2 auront lieu dans n'importe quel ordre,
-act1 $\oplus$ act2: signifie qu'une des deux occurrences seulement aura lieu (ou exclusif).

Dans certains cas, il s'avère intéressant d'utiliser un action spéciale appelée DAC (dummy action) afin de simplifier des expressions complexes. Par exemple:

"(a;b;c;) ⊕ (a;c;)" peut être simplifiée en "a;(b DAC);c".

d) **Capability**

Les contraintes de cette sous-catégorie permettent d'exprimer la capacité qu'a un agent à exécuter une action. Pour cela, nous utiliserons une autre extension à la logique classique du premier ordre. Ainsi, trois nouveaux connecteurs rendront possible l'expression du concept de permission associé à un agent: il s'agit des opérateurs "**obligation**" (**O**), "**forbidden**" (**F**) et "**exclusive obligation**" (**XO**).

La syntaxe pour une "obligation" est la suivante: **O(<int-action> / <situation>)**. Celle-ci exprime que si les circonstances décrites dans <situation> sont rencontrées, alors l'action interne <int-action> sera d'office réalisée. Par contre, si ces circonstances ne sont pas rencontrées, l'action pourra, selon les cas, être ou ne pas être réalisée. Ces circonstances sont des conditions sur la structure d'état.

La syntaxe pour une "interdiction" est la suivante: **F(<int-action> / <situation>)**. Celle-ci exprime que si les circonstances décrites dans <situation> sont rencontrées, alors l'action interne <int-action> ne sera en aucun cas réalisée. Par contre, si ces circonstances ne sont pas rencontrées, l'action pourra, selon les cas, être ou ne pas être réalisée. Ces circonstances sont des conditions sur la structure d'état.

La syntaxe pour une "obligation exclusive" est la suivante: **XO(<int-action> / <situation>)**. Cette expression est un raccourci d'écriture exprimant O(<int-action> / <situation>) ∧ F(<int-action> / ¬ <situation>).

II.2.3.3. Les contraintes de coopération (Cooperation Constraints).

II.2.3.3.1. Définitions.

Les contraintes de coopération sont utilisées pour décrire la façon dont l'agent interagit avec son environnement, c'est-à-dire pour décrire les circonstances par lesquelles:

-cet agent peut percevoir les actions exécutées par d'autres agents (sous-catégorie **Action Perception**),

-cet agent peut avoir accès, en partie ou en totalité, à l'état des autres agents (sous-catégorie **State Perception**),

-d'autres agents peuvent percevoir les actions exécutées par cet agent (sous-catégorie **Action Information**),

-d'autres agents peuvent avoir accès, en partie ou en totalité, à l'état de cet agent (sous-catégorie **State-Information**).

Ces contraintes permettent ainsi à l'analyste d'ajouter une dimension dynamique aux concepts d'importation et d'exportation définis dans la partie déclaration de la spécification.

II.2.3.3.2. Syntaxe.

a) **Action Perception.**

Les perceptions d'actions sont décrites en utilisant les connecteurs logiques "**knowledge**" (**K**), "**ignorance**" (**I**) et "**exclusive knowledge**" (**XK**).

La syntaxe pour un "knowledge" est la suivante: **K(<ext-action> / <situation>)**. Celle-ci exprime que si les circonstances décrites dans <situation> sont rencontrées, alors les occurrences d'un action externe <ext-action> seront d'office perçues par l'agent. Par contre, si ces circonstances ne sont pas rencontrées, les occurrences pourront, selon les cas, être ou ne pas être perçues par l'agent. Ces circonstances sont des conditions sur la structure d'état.

La syntaxe pour une "ignorance" est la suivante: **I(<ext-action> / <situation>)**. Celle-ci exprime que si les circonstances décrites dans <situation> sont rencontrées, alors les occurrences d'une action externe <ext-action> ne seront en aucun cas perçues par l'agent. Par contre, si ces circonstances ne sont pas rencontrées, les occurrences pourront, selon les cas, être ou ne pas être perçues par l'agent. Ces circonstances sont des conditions sur la structure d'état.

La syntaxe pour un "knowledge exclusif" est la suivante: **XK(<ext-action> / <situation>)**. Cette expression est un raccourci d'écriture exprimant K(<ext-action> / <situation>) $\wedge$ I(<ext-action> / $\neg$ <situation>).

La règle par défaut est que les actions importées disponibles peuvent être perçues quelle que soit la situation.

b) State perception.

Les perceptions d'état sont décrites en utilisant les mêmes connecteurs logiques que ceux des perceptions d'actions c.-à-d. K, I, et XK.

La règle par défaut est que les composantes d'état importées disponibles peuvent être perçues, quelle que soit la situation.

c) Action Information.

Cette contrainte est également décrite en utilisant les connecteurs logiques K, I et XK.

La syntaxe pour un "knowledge" est la suivante: **K(<int-action>.<agent> / <situation>)**. Celle-ci exprime que si les circonstances décrites dans <situation> sont rencontrées, alors les occurrences d'un action interne <int-action> seront d'office rendues disponibles à l'agent <agent>. Par contre, si ces circonstances ne sont pas rencontrées, les occurrences pourront, selon les cas, être ou ne pas être rendues disponibles à l'agent.

La syntaxe pour une "ignorance" est la suivante: **I(<int-action>.<agent> / <situation>)**. Celle-ci exprime que si les circonstances décrites dans <situation> sont rencontrées, alors les occurrences d'une action interne <int-action> ne seront en aucun cas rendues disponibles à l'agent. Par contre, si ces circonstances ne sont pas rencontrées, les occurrences pourront, selon les cas, être ou ne pas être rendues disponibles à l'agent.

La syntaxe pour un "knowledge exclusif" est la suivante: **XK(<int-action>.<agent> / <situation>)**. Celle-ci exprime que si les circonstances décrites dans <situation> sont rencontrées, alors les occurrences d'une action interne <int-action>

seront rendues disponibles à l'agent. Par contre, si ces circonstances ne sont pas rencontrées, les occurrences ne pourront en aucun cas lui être disponibles. Cette expression est un raccourci exprimant K(<int-action>.<agent> / <situation>) ∧ I(<int-action>.<agent> / ¬ <situation>).

La règle par défaut est que toutes les actions exportées peuvent être vues par tous les agents pour lesquels ces actions sont exportées, quelque soit la situation.

d) State Information.

Les contraintes sont également décrites en utilisant les connecteurs logiques K, I et XK.

La règle par défaut est que toutes les composantes d'état exportées peuvent être vues par tous les agents pour lesquels ces composantes sont exportées, quelque soit la situation.

II.2.4. Commentaires sur les contraintes.

A ce stade, des commentaires concernant la concurrence, le non-déterminisme, le contrôle fin de la visibilité et la fiabilité des agents décrits par les contraintes, s'imposent:

-La concurrence et le non-déterminisme.

Comme la plupart des autres langages, le langage ALBERT peut décrire les systèmes de manière opérationnelle. Mais l'intérêt réel qu'il présente sur les autres langages, est qu'il peut également les décrire de manière déclarative. Les contraintes de causalité vues au II.2.3.2. b sont un bon exemple de description déclarative. Une telle description déclarative est avantageuse lorsqu'on veut décrire un système qui présente des propriétés de concurrence. En effet, elle permet d'éviter les problèmes relatifs au "deadlocks" dans la phase de spécification des besoins, en ce sens que la contrainte de causalité ainsi décrite suffit à exprimer qu'on désire les exclure du système. C'est dans les phases ultérieures que des solutions concernant ces problèmes seront abordées.

La description déclarative de ces contraintes permet également d'exprimer des propriétés non-déterministes. Pendant la période définie depuis l'occurrence d'une action jusqu'à l'occurrence de l'action dont l'exécution dépend de la première (via la contrainte de causalité), le non-déterminisme s'applique. Cela signifie que pendant cette période, la contrainte de causalité n'implique nullement que d'autres actions ne puissent survenir.

-Le contrôle fin de la visibilité.

Le principe de visibilité, consistant au fait que les agents peuvent accéder aux informations d'autres agents (composantes d'état), est très élaboré dans le langage ALBERT. Les mécanismes décrits à la figure 2.9. permettent un contrôle fin et précis de ce principe de visibilité. Le premier mécanisme consiste à exporter les informations d'un agent vers un autre. Le second mécanisme permet, via les contraintes "State Information", de déterminer les conditions d'exportation, lesquelles présentent des propriétés dynamiques. Le troisième mécanisme permet, via les contraintes "State Perception" de déterminer les conditions de perception, lesquelles présentent également des propriétés dynamiques. Notons que les deux derniers mécanismes décrits ne peuvent être réalisés qu'à partir du moment ou le premier mécanisme décrit aura été mis

en place. Ces trois mécanismes se comportent comme des filtres successifs sur des "canaux" transportant les informations entre agents.

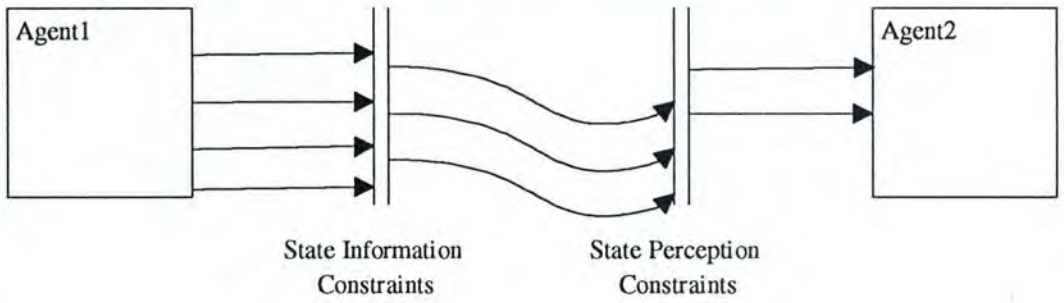

Figure 2.9. Illustration du contrôle fin de la visibilité.

-La fiabilité des agents.

Un mécanisme à trois niveaux semblable à celui qui vient d'être exposé est fournit en langage ALBERT pour traiter le flux de contrôle entre agents. Ce flux de contrôle permet à un agent d'avoir la possibilité de faire savoir à un autre agent qu'il a exécuté une action. Cette possibilité est fonction des contraintes "Action Information", lesquelles sont dynamiques. Notons que la perception de cette action dépend des contraintes "Action Perception", lesquelles sont également dynamiques. Il est dès lors possible à l'analyste de décrire la fiabilité des agents, correspondant à l'obligation pour chaque agent de faire savoir qu'il a exécuté une action d'une part, et à l'obligation pour chaque agent de percevoir les actions exécutées d'autre part.

Chapitre III:

Spécification ALBERT d'un logiciel de messagerie électronique.X.400, EAN.

III.I. Introduction.

Dans ce chapitre, nous spécifierons en langage ALBERT les besoins concernant un système de messagerie électronique X.400-84, en utilisant comme cas la version 2.1 du logiciel EAN. Nous attirons l'attention du lecteur sur le fait que le but du travail n'est pas de **tout** décrire mais bien d'exposer les spécifications de base nécessaires pour aborder au chapitre VI les problèmes liés à la sécurité du système et que l'on spécifiera un serveur de sécurité. C'est ainsi que par exemple nous ne décrirons pas les adresses X.400: le langage ALBERT fournit un identifiant pour chaque instance d'une classe d'agents et un message pourrait donc être délivré à un UA lorsque le champ recip (c.-à-d. la liste des destinataires) de l'enveloppe du message (cf. annexe 1.) contient les identifiants de tous les UA auxquels le message doit être délivré.

L'avantage que présente le langage ALBERT sur le langage ASN.1 (Abstract Syntax Notation 1) utilisé pour spécifier les recommandations X.400 est qu'il permet une simplification dans la façon d'aborder un problème en ce sens qu'on ne doit pas nécessairement tenir compte de tous les éléments de celui-ci.

III.2. Spécification en ALBERT de EAN.

III.2.1. Déclaration de la hiérarchie des agents.

EAN est un système de messagerie électronique basé sur les recommandations X.400 (X.400-84). Il est constitué de trois grands modules:

-le **User Service (US)**,
-le **Message Transfer Service (MTS)**,
-le **Directory Service.(DS)**.

Les deux premiers modules cités fournissent les principaux services décrits dans le chaptre I, services qui étaient offerts respectivement par la couche UAL (primitives de service offertes à l'UA) et la couche MTL (primitives de service offertes au MTA).

Le User Service est constitué d'un ensemble de programmes correspondant aux UAs vus au chapitre I. Ce US est un interface entre les utilisateurs et le système de transfert des messages. Le Directory Service permet d'obtenir des renseignements divers au sujet d'un utilisateur. Il peut par exemple traduire un O/R name en un O/R address pour un utilisateur donné. C'est dans la série de recommandations X.500 que ce concept est né. Cependant, EAN a défini un service d'annuaire alors que les normes n'étaient pas encore bien établies, et l'a adapté à ses besoins de l'époque, en ne tenant pas compte de la possibilité que possède ce service à stocker des clés publiques. Ce service nous intéressera plus particulièrement au chapitre VI, car il nous permettra de définir un dialogue entre le serveur de sécurité (défini au chapitre VI), l'UA et le DS.

La figure 3.1. nous décrit la hiérarchie complète des agents. Repartons de nos trois catégories d'agents, citées ci-avant, pour exposer celle-ci: les agents User-service et MTS, qui sont des agents composés constitués d'un ensemble d'agents simples, respectivement UA et MTA, et l'agent DS, qui est un agent simple.

Ces trois catégories d'agents agissent chacune à leur manière et selon leur rôle qui leur est propre pour contribuer à la réalisation de la fonction de l'agent composé MHS, leur supérieur hiérarchique. De la même manière, l'agent MHS et l'ensemble des agents User vont agir ensemble pour permettre à l'agent composé MHE de réaliser sa fonction.

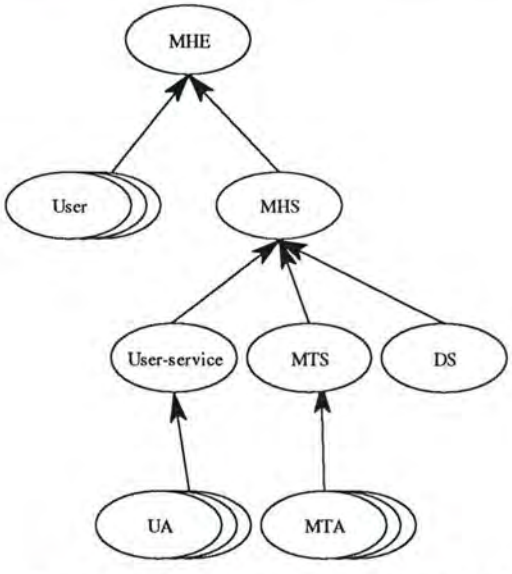

Figure 3.1.: Hiérarchie des agents du logiciel EAN.

Décrivons à présent, dans les paragraphes qui suivent, les différents agents simples et classe d'agents repris à la figure 3.1.

III.2.2. Description de l'agent "User".

a) Déclaration de l'agent.

La première phase de la spécification de l'agent est la déclaration graphique en langage ALBERT des composantes d'état et des actions dont il a la responsabilité d'une part, et des composantes d'état et des actions qui lui sont importées d'autre part (fig.3.2.). Le tableau 3.1. de l'annexe 2 donne la liste des composantes d'états et des actions qui sont sous la responsabilité de l'agent **User**, avec une brève description de celles-ci. Quant à la description des éléments importés, le lecteur peut se référer au tableau de description de l'agent qui exporte cet élément, lequel se trouve également à l'annexe 2. Les définitions des types abstraits de données se trouvent à l'annexe 3.

Bien que nous n'ayons pas encore décrit les autres agents du système, nous pouvons constater, à la figure 3.2., que l'utilisateur ne dialogue qu'avec le UA, et jamais avec un autre agent. Dans le chapitre VI, nous évoquerons le fait que l'utilisateur puisse dialoguer avec n'importe quel élément du système par simple imposture.

b) Spécification des contraintes.

La figure 3.3. reprend la spécification des contraintes.

User AGENT CONSTRAINTS

BASIC CONSTRAINTS

Derived Components

Initial Conditions

LOCAL CONSTRAINTS

States Behaviour

Effects Of Actions

*
ua.Get-can-session-conf(-,ok): UA-already-used=TRUE
*
ua.Stop-can-session-conf(-): UA-already-used=FALSE

Causality

Capability

* Cap1
F(**Message-compose-req(ua)** / ¬ UA-already-used ∨ ua <> UA-accessible)
* Cap2
F(**Message-submission-req(ua)** / ¬ UA-already-used ∨ ua <> UA-accessible)
* Cap3
F(**Message-wait-req(ua)** / ¬ UA-already-used ∨ ua <> UA-accessible)
* Cap4
F(**Message-delete-req(ua,-)** / ¬ UA-already-used ∨ ua <> UA-accessible)
* Cap5
F(**Message-read-req(ua,-)** / ¬ UA-already-used ∨ ua <> UA-accessible)
* Cap6
F(**Get-can-session-req(ua)** / UA-already-used ∨ ua <> UA-accessible)
* Cap7
F(**UA-drop-req(ua)** / ¬ UA-already-used ∨ ua <> UA-accessible)
* Cap8
F(**UA-find-req(ua,-)** / ¬ UA-already-used)¬ UA-already-used ∨ ua <> UA-accessible)
* Cap9
F(**UA-register-req(ua,-)** / ¬ UA-already-used)¬ UA-already-used ∨ ua <> UA-accessible)
* CAp10
F(**Stop-can-session-req(ua)** / ¬ UA-already-used ∨ ua <> UA-accessible)
* CAp11
F(**Change-folder-req(ua)** / ¬ UA-already-used ∨ ua <> UA-accessible)

COOPERATION CONSTRAINTS

Figure 3.3.: Spécification des contraintes pour l'agent UA en ALBERT (suite).

Action Perception

* AP1
  XK(ua.Message-submission-conf(user) / UA-already-used ∧ UA-accessible=ua ∧ user=self)
* AP2
  XK(ua.Message-compose-conf(user) / UA-already-used ∧ UA-accessible=ua ∧ user=self)
* AP3
  XK(ua.Message-read-conf(user,-) / UA-already-used ∧ UA-accessible=ua ∧ user=self)
* AP4
  XK(ua.Message-delete-conf(user,-) / UA-already-used ∧ UA-accessible=ua ∧ user=self)
* AP5
  XK(ua.Message-wait-conf(user) / UA-already-used ∧ UA-accessible=ua ∧ user=self)
*AP6
  XK(ua.UA-drop-conf(ua) / UA-already-used ∧ UA-accessible=ua ∧ user=self)
* AP7
  XK(ua.UA-find-conf(ua,-) / UA-already-used ∧ UA-accessible=ua ∧ user=self)
* AP8
  XK(ua.UA-register-conf(ua,-) / UA-already-used ∧ UA-accessible=ua ∧ user=self)
* AP9
  XK(ua.Get-can-session-conf(user,-) / ¬ UA-already-used ∧ UA-accessible=ua ∧ user=self)
* AP10
  XK(ua.Stop-can-session-conf(user) / UA-already-used ∧ UA-accessible=ua ∧ user=self)
* AP11
  XK(ua.Change-folder-conf(user,-) / UA-already-used ∧ UA-accessible=ua ∧ user=self)

State Perception

* SP1
  XK(ua.Messages-folders / UA-already-used ∧ UA-accessible=ua)
* SP2
  XK(ua.Stored-messages / UA-already-used ∧ UA-accessible=ua)
* SP3
  XK(ua.Stored-reports / UA-already-used ∧ UA-accessible=ua)
* SP4
  XK(ua.Draft-messages / UA-already-used ∧ UA-accessible=ua)
* SP5
  XK(ua.Draft-pos / UA-already-used ∧ UA-accessible=ua)
* SP6
  XK(ua.Current-folder / UA-already-used ∧ UA-accessible=ua)

Action Information

* AI1
  XK(Message-submission-req(ua').ua / UA-already-used ∧ UA-accessible=ua ∧ ua=ua')
* AI2
  XK(Message-compose-req(ua').ua / UA-already-used ∧ UA-accessible=ua ∧ ua=ua')
* AI3
  XK(Message-read-req(ua',-).ua / UA-already-used ∧ UA-accessible=ua ∧ ua=ua')
* AI4
  XK(Message-delete-req(ua',-).ua / UA-already-used ∧ UA-accessible=ua ∧ ua=ua')
* AI5
  XK(Message-wait-req(ua').ua / UA-already-used ∧ UA-accessible=ua ∧ ua=ua')
* AI6
  XK(UA-drop-req(ua').ua / UA-already-used ∧ UA-accessible=ua ∧ ua=ua')
* AI7
  XK(UA-find-req(ua',-).ua / UA-already-used ∧ UA-accessible=ua ∧ ua=ua')
* AI8
  XK(UA-register-req(ua',-).ua / UA-already-used ∧ UA-accessible=ua ∧ ua=ua')

State Information

Figure 3.3.: Spécification des contraintes pour l'agent User en ALBERT (suite).

Remarques:

La demande de connexion d'un utilisateur au logiciel EAN a pour effet de placer la valeur de l'instance UA-already-used à TRUE. A partir de cet instant, n'importe quelle action peut être déclenchée par l'agent User (hormis une nouvelle demande de session avec EAN puisqu'un utilisateur ne peut accéder qu'une seule fois au logiciel). A chacune de ces actions correspondra un paramètre de type UA. Au cours de la vie de l'agent, ce paramètre prendra toujours la même valeur, à savoir celle contenue dans l'instance UA-accessible. Cela a pour effet d'identifier l'UA pour lequel cette requête est formulée. Actuellement, aucune action permettant à l'utilisateur de modifier la valeur de UA-accessible n'est prévue. Cela évite, en théorie, toute forme d'imposture. Cependant, dans la pratique, on est malheureusement amené à constater qu'on n'est pas totalement à l'abri de ce risque. La spécification décrite ici est en fait une spécification idéale et nous verrons au chapitre VI que l'agent User peut être capable d'actions mal-intentionnées.

III.2.3. Description de l'agent "UA".

a) Déclaration de l'agent.

Le tableau 3.2. de l'annexe 2 donne la liste des composantes d'état et des actions qui sont sous la responsabilité de l'agent UA, avec une brève description de celles-ci. Quant à la description des éléments importés, le lecteur peut se référer au tableau de l'agent exportant se trouvant également à l'annexe 2. Les définitions des types abstraits de données se trouvent à l'annexe 3.

Les figures 3.4. et 3.5. représentent la déclaration graphique de l'agent UA en langage ALBERT. Les actions de type ACT-Conf (ex: SUBMIT-Req, DS-FIND-Req, LOGON-Req, ...) sont soit des actions représentant des primitives de service X.400 (ex: SUBMIT-Req, LOGON-Req, ...), soit des actions exécutées par une procédure logicielle et pouvant être perçues par une autre procédure logicielle (ex: DS-FIND_Req, ...).

Les actions de type Act-req ou Act-conf (ex: Get-ean-session-conf, ...) sont des actions exécutées par un utilisateur ou par une procédure logicielle et perçues respectivement par une procédure logicielle ou un utilisateur.

Si à un instant donné, aucune session EAN ne relie un UA à un utilisateur, la valeur de l'instance UA-stat est à IDLE. Ni message ni rapport de livraison ne peuvent être délivrés ou notifiés à cet UA.

La description en langage ALBERT de la structure de rangement des messages dans le logiciel EAN est la suivante: une population appelée Message-folders est constituée d'un ensemble d'espaces de rangement (folder); à chaque espace de rangement est associé un tableau reprenant les identifiants des messages qu'il contient. Les messages proprement dits, quant à eux, sont stockés dans la population Stored-messages.

C'est grâce à l'instance User-in-session que l'UA peut répondre à l'utilisateur (Actions Act-conf) qui lui a fait une requête (Actions Act-req).

b) Spécification des contraintes.

La figure 3.6. reprend la spécification des contraintes.

UA AGENT CONSTRAINTS

BASIC CONSTRAINTS

Derived Components

Initial Conditions

Stored-messages[-] = UNDEF
Messages-folders [-] = UNDEF
UA-stat = IDLE
Current-folder="in"
Action-in-work = FALSE
Logon = FALSE
Deliver-and-notify = FALSE

LOCAL CONSTRAINTS

States Behaviour

* Pour tout message contenu dans un folder et créé par cet UA, il (son adresse) doit apparaître dans la liste des destinataires (recip) dans l'enveloppe du message.

Message-folders[i] = tm ∧ Message-folders[i] <> UNDEF ∧ tm[j] <> UNDEF ∧
In-out(UA-Header[tm[j]] = 'in'
⇒ self ∈ Recip(Enveloppe(Stored-Messages[tm[j]]))

* Pour tout message contenu dans un folder et créé par cet UA, il (son adresse) doit apparaître expéditeur (origin) dans l'enveloppe du message.

Message-folders[i] = tm ∧ Message-folders[i] <> UNDEF ∧ tm[j] <> UNDEF ∧
In-out(UA-header[tm[j]] = 'out'
⇒ Origin(Enveloppe(Stored-Messages[tm[j]])) = self

* Pour un message manipulé sur le moment même et contenu dans Stored-messages, le status de celui-ci doit être à "Draft".

Message-folders[i] = tm ∧ Message-folders[i] <> UNDEF ∧ tm[j] <> UNDEF ∧
Draft-message = Stored-messages[tm[j]]
⇒ Draft(Status(UA-header[tm[j]])) = TRUE
*
Message-folders[i] = tm ∧ Message-folders[i] <> UNDEF ∧ tm[j] <> UNDEF ∧
Draft-message <> Stored-messages[tm[j]]
⇒ Draft(Status(UA-header[tm[j]])) = FALSE

Figure 3.6.: Spécification des contraintes pour l'agent UA en ALBERT.

Effects Of Actions

* EA1: Quand un message est délivré, il est stocké dans le folder in-folder de l'attribut UA-profile de User-in-session

mta.DELIVER-Ind(ua,m): tm[i] = mid
with i = Max(Message-folders["in"]) ∧ mid = Messid(Enveloppe(m)) ∧
Message-folders["in"]) = tm
New(Status(UA-header[mid]))) = TRUE
Read(Status(UA-header[mid])) = FALSE
Draft(Status(UA-header[mid])) = FALSE
Rprt(Status(UA-header[mid])) = FALSE
Del(Status(UA-header[mid])) = FALSE
In-out(UA-header[mid]) = 'in'
Origin(UA-header[mid]) = Origin(Enveloppe(m))
Recip(UA-header[mid]) = self
Date(UA-header[mid]) = Date(Enveloppe(m))
Stored-messages[mid] = m

* EA2: Notifcation

mta.NOTIFY-Ind(ua,r): Stored-reports[m] = r
Rprt(Status(UA-header[mid])) = TRUE

* EA3:

mta.SUBMIT-Conf(mta,m) with Draft-pos = UNDEF:
tm[Draft-pos] = mid
with Draft-pos=Max(Message-folders[Current-folder]) ∧
mid = Messid(Enveloppe(Draft-message)) ∧
Message-folders["in"] = tm
New(Status(UA-header[mid])) = FALSE
Read(Status(UA-header[mid])) = TRUE
Draft(Status(UA-header[mid])) = TRUE
Rprt(Status(UA-header[mid])) = FALSE
Del(Status(UA-header[mid])) = FALSE
In-out(UA-header[mid]) = 'out'
Origin(UA-header[mid]) = self
Recip(UA-header[mid]) =Recip(Enveloppe(Draft-message))
Date(UA-header[mid]) = Date(Enveloppe(Draft-message))
Stored-messages[mid] = m
Action-in-work = FALSE
Deliver-and-notify = TRUE

* EA4

mta.SUBMIT-Conf(ua,m) with Draft-pos <> UNDEF:
tm[Draft-pos] = mid
with mid = Messid(Draft-message) ∧
Message-folders["in"] = tm
New(Status(UA-header[mid])) = FALSE
Read(Status(UA-header[mid])) = TRUE
Draft(Status(UA-header[mid])) = TRUE
Rprt(Status(UA-header[mid])) = FALSE
Del(Status(UA-header[mid])) = FALSE
In-out(UA-header[mid]) = 'out'
Origin(UA-header[mid]) = self
Recip(UA-header[mid]) =Recip(Enveloppe(Draft-message))
Date(UA-header[mid]) = Date(Enveloppe(Draft-message))
Stored-messages[mid] = m
Action-in-work = FALSE
Deliver-and-notify = TRUE

* EA5

mta.LOGON-Conf(mta): Logon = TRUE

* EA6

mta.LOGOFF-Conf(mta): Logon = FALSE
Deliver-and-notify = FALSE

Figure 3.6.: Spécification des contraintes pour l'agent UA en ALBERT (suite).

```
* EA7
        Message-wait-conf(-,res) with Draft-pos = UNDEF ∧ res = ok:
                tm[Draft-pos] = mid
                        with Draft-pos=Max(Message-folders[Current-folder]) ∧
                        mid = Messid(Enveloppe(Draft-message)) ∧
                        Message-folders[Current-folder] = tm
                Stored-messages[mid] = m
                Action-in-work = FALSE
* EA8
        Message-wait-conf(-,res) with Draft-pos <> UNDEF ∧ res = ok:
                tm[Draft-pos] = mid
                        with mid = Messid(Enveloppe(Draft-message))
                        Message-folders[Current-folder] = tm
                Stored-messages[mid] = m
                Action-in-work = FALSE
* EA9
        Message-delete-conf(-,i,res) with res = ok::
                Delete(Status(UA-header[tm[i]])) = TRUE
                Action-in-work = FALSE
                with Message-folders[Current-folder] = tm
* EA10
        Message-read-conf(-,i,res) with res = ok
                tm[Draft-pos] = mid
                Read(Status(UA-header[mid])) = TRUE
                Draft-pos = i
                Draft-message = Stored-messages[mid]
                Action-in-work = FALSE
                        with Message-folders[Current-folder] = tm
* EA11
        Stop-can-session-conf(-,res) with res = ok: UA-stat = IDLE
                                                    User-in-session = UNDEF
* EA12
        Get-can-session-conf(us,resp) with resp=TRUE: UA-stat = NOT IDLE
                                                      Action-in-work = FALSE
                                                      User-in-session = us
* EA13
        Begin-get-messages: Deliver-and-notify = TRUE
* EA14
        Finish-get-messages: Deliver-and-notify = FALSE
* EA15
        Change-folder-conf(-,f): Current-folder=f
* EA16
        user.Message-submission-req(-):
                Action-in-work = TRUE
                Draft(Status(UA-header[tm[Draft-pos]])) = TRUE
                        with Draft-pos <> UNDEF
                        Message-folders[Current-folder] = tm
* EA17
        user.Message-wait-req(-): Action-in-work = TRUE
* EA18
        user.Message-delete-req(-,i): Action-in-work = TRUE
* EA19
        user.Message-read-req(-,i): Action-in-work = TRUE
* EA21
        user.Stop-can-session-req(-): Action-in-work = TRUE
* EA22
        user.Get-can-session-req(-,-): Action-in-work = TRUE
* EA23
        user.Change-folder-req(-,-): Action-in-work = TRUE
```

Figure 3.6.: Spécification des contraintes pour l'agent UA en ALBERT (suite).

Causality

* C1: Quand un utilisateur demande pour que l'UA soumette un message, il sera averti du résultat de cette demande.

user.Message-submission-req(-) $\xrightarrow{\Diamond}$ Message-submission-conf(-,-)

* C2: Quand un message a été correctement soumis au MTA, l'utilisateur reçoit la confirmation que tout c'est correctement passé.

user.Message-submission-req(ua);
LOGON-Req(mta,ap);mta.LOGON-Conf(ua,ok);
SUBMIT-Req(mta,Draft-message);mta.SUBMIT-Conf(ua,m);
LOGOFF-Req(mta);mta.LOGOFF-Conf(ua) $\xrightarrow{\Diamond}$ Message-submission-conf(ua,ok)

* C3

user.Message-compose-req(-) $\xrightarrow{\Diamond}$ Message-compose-conf(-,-)

* C4

user.Message-wait-req(-) $\xrightarrow{\Diamond}$ Message-wait-conf(-,-)

* C5

user.Message-delete-req(-,-) $\xrightarrow{\Diamond}$ Message-delete-conf(-,-,-)

* C6

user.Message-read-req(-,-) $\xrightarrow{\Diamond}$ Message-read-conf(-,-,-)

* C7.

user.Get-can-session-req(-,-) $\xrightarrow{\Diamond_{SI'}}$ Get-can-session-conf(-,-)

* C8: Quand un utilisateur obtient l'autorisation de se connecter, il peut obtenir les messages qui attendait à être délivré ou les rapports à être notifié à partir du MTA connu dans MTA-accessible.

user.Get-can-session-req(-,-);Get-can-session-conf(-,-);
LOGON-Req(mta,ap);mta.LOGON-Conf(ua,ok);
Begin-get-messages;Finish-get-messages;
LOGOFF-Req(mta);mta.LOGOFF-Conf(ua) $\xrightarrow{\Diamond}$ Get-can-session-conf(-,-)

* C9

user.Stop-can-session-req(-) $\xrightarrow{\Diamond}$ Stop-can-session-conf(-,-)

* C10

user.Change-folder-req(-,-) $\xrightarrow{\Diamond}$ Change-folder-conf(-,-)

Capability

* Cap1

F(**Message-compose-conf(us,-)** / User-in-session <> us ∨ UA-stat = IDLE)

* Cap2

F(**Message-delete-conf(us,i,-)** / User-in-session <> us ∨ UA-stat = IDLE ∨ Current-folder = UNDEF ∨
tm[i] = UNDEF with Message-folders[Current-folder] = tm)

* Cap3

F(**Message-read-conf(us,i,-)** / User-in-session <> us ∨ UA-stat = IDLE ∨ Current-folder = UNDEF ∨
tm[i] = UNDEF with Message-folders[Current-folder] = tm)

* Cap4

F(**Message-submission-conf(us,-)** / User-in-session <> us ∨ UA-stat = IDLE ∨ Current-folder = UNDEF ∨
Draft-message <> UNDEF)

* Cap5

F(**Message-wait-conf(us,-)** / User-in-session <> us ∨ UA-stat = IDLE ∨ Current-folder = UNDEF ∨
Draft-message <> UNDEF)

* Cap6

F(**Change-folder-conf(us,f)** / User-in-session <> us ∨ UA-stat = IDLE ∨ Message-folder[f]=UNDEF)

* Cap7

F(**LOGON-Req(mta,ap)** / UA-stat = IDLE ∨ Logon ∨ mta <> MTA-accessible)

* Cap8

F(**SUBMIT-Req(mta)** / UA-stat = IDLE ∨ ¬ Logon ∨ mta <> MTA-accessible)

Figure 3.6.: Spécification des contraintes pour l'agent UA en ALBERT (suite).

* Cap9

F(**LOGOFF-Req(mta)** / UA-stat = IDLE ∨ ¬ Logon ∨ mta <> MTA-accessible)

* Cap10

F(**Begin-get-messages** / UA-stat = IDLE ∨ Logon)

* Cap12

F(**Finish-get-messages** / UA-stat = IDLE ∨ Logon)

COOPERATION CONSTRAINTS

Action Perception

* AP1: Un UA ignore la perception de la livraison de message s'il n'est pas un des destinataires ou si UA est inoccupé ou si l'UA n'est pas en condition pour se faire délivrer des messages.

I(**mta.DELIVER-Ind(ua,m)** / Recip(Enveloppe(m))<>ua ∨ ua<>self ∨ UA-stat=IDLE ∨ ¬ Logon ∨
¬ Deliver-and-notify ∨ mta <> MTA-accessible)

* AP2: Un UA ignore la perception de la notification d'un rapport de livraison/non-livraison s'il n'est pas le destinataire du rapport ou s'il n'est pas l'expéditeur ou si UA est inoccupé ou si l'UA n'est pas en condition pour se faire délivrer des messages.

I(**mta.NOTIFY-Ind(ua,r,m)** / Recip(Enveloppe(r)) <> ua ∨ Origin(Enveloppe(m) <> ua ∨
m ∈ Message-folders["in"] ∨ Rprt(Status(UA-header[mid])) = FALSE ∨ ua <> self ∨
UA-stat = IDLE ∨ mid <> Messid(Enveloppe(m)) ∨ mta <> MTA-accessible)

* AP3

I(**user.Stop-can-session-req(ua)** / ua <> self ∨ user <> User-in-session ∨ ¬ Action-in-work ∨
UA-stat = IDLE)

* AP4

I(**user.Message-compose-req(ua)** / ua <> self ∨ user <> User-in-session ∨ ¬ Action-in-work ∨
UA-stat = IDLE)

* AP5

I(**user.Message-submission-req(ua,-)** / ua <> self ∨ user <> User-in-session ∨ ¬ Action-in-work ∨
UA-stat = IDLE)

* AP6

I(**user.Message-wait-req(ua)** / ua <> self ∨ user <> User-in-session ∨ ¬ Action-in-work ∨
UA-stat = IDLE)

* AP7

I(**user.Message-delete-req(ua,-)** / ua <> self ∨ user <> User-in-session ∨ ¬ Action-in-work ∨
UA-stat = IDLE)

* AP8

I(**user.Message-read-req(ua,-)** / ua <> self ∨ user <> User-in-session ∨ ¬ Action-in-work ∨
UA-stat = IDLE)

* AP8

I(**mta.LOGON-Conf(ua,ap)** / UA-stat = IDLE ∨ ¬ Logon ∨ ¬ Deliver-and-notify ∨
mta <> MTA-accessible ∨ ua <> self)

* AP9

I(**mta.SUBMIT-Conf(ua)** / ∨ UA-stat = IDLE ∨ ¬ Logon ∨ mta <> MTA-accessible ∨ ua <> self)

* AP10

I(**LOGOFF-Req(mta)** / UA-stat = IDLE ∨ ¬ Logon ∨ mta <> MTA-accessible ∨ ua <> self)

State Perception

Action Information

* AI1

XK(**Get-can-session-conf(us,-,-).user** / User-in-session = us ∧ us = user ∧ us=self)

* AI2

XK(**Message-compose-conf(us,-).user** / User-in-session = us ∧ us = user ∧ us=self ∧
Current-folder <> UNDEF)

Figure 3.6.: Spécification des contraintes pour l'agent UA en ALBERT (suite).

* AI3

XK(**Message-delete-conf(us,i,-).user** / User-in-session = us ∧ us = user ∧ us=self ∧
Current-folder <> UNDEF ∧ tm[i] = UNDEF with Message-folders[Current-folder] = tm)

* AI4

XK(**Message-read-conf(us,i,-).user** / User-in-session = us ∧ us = user ∧ us=self ∧
Current-folder <> UNDEF ∧ tm[i] = UNDEF with Message-folders[Current-folder] = tm)

* AI5

XK(**Message-submission-conf(us,-).user** / User-in-session = us ∧ us = user ∧ us=self ∧
Current-folder <> UNDEF)

* AI6

XK(**SUBMIT-Req(mta,m).mta'** / Current-folder <> UNDEF ∧ mta = MTA-accessible ∧ mta = mta' ∧ Logon)

* AI7

XK(**LOGON-Req(ua).mta'** / mta = MTA-accessible ∧ mta = mta' ∧ ¬Logon)

* AI8

XK(**LOGOFF-Req(ua)**.mta' / mta = MTA-accessible ∧ mta = mta' ∧ ¬Logon)

* AI9

XK(**Message-wait-conf**(us,-).user / User-in-session = us ∧ us = user ∧ us=self ∧
Current-folder <> UNDEF)

State Information

* SI1

XK(**Message-folders.user** / User-in-session = user ∧ user=self)

* SI2

XK(**Stored-messages.user** / User-in-session = user ∧ user=self)

* SI3

XK(**Stored-reports.user** / User-in-session = user ∧ user=self)

* SI4

XK(**Draft-message.user** / User-in-session = user ∧ user=self)

* SI5

XK(**Draft-pos.user** / User-in-session = user ∧ user=self)

Figure 3.6.: Spécification des contraintes pour l'agent UA en ALBERT (suite).

Remarques:

Lorsque l'utilisateur désire soumettre un message à l'UA, il le fait via la requête Message-submission-req. Les contraintes de causalité impliquent deux cas de figures:

-soit l'action interne Message-submission-conf révèlera que le message a été effectivement soumis (ok). Dans ce cas, depuis le moment où la requête a été réalisée par l'utilisateur jusqu'à la confirmation de celle-ci par l'UA, une série de protocoles auront été échangés entre l'UA et le MTA, afin que le message soit transmis à ce dernier. Ces protocoles sont caractérisés par les actions LOGON-Req, LOGON-Conf, SUBMIT-Req, SUBMIT-Conf, LOGOFF-Req et LOGOFF-Conf,

-soit l'action interne Message-submission-conf révèlera que le message n'aura pas été soumis (nok). Dans ce cas la succession des actions décrites ci-avant n'aura jamais été réalisée entièrement.

Notons également que l'action SUBMIT-Conf de confirmation, laquelle sera décrite par la suite, est exécutée par l'agent MTA et porte sur un message qui n'est pas forcément celui qui lui a été transmis par l'action SUBMIT-Req. Ainsi, un imposteur suffisamment habile, serait capable de modifier le message envoyé par un utilisateur hônnete et ce serait ce message modifié qui serait effectivement transmis et confirmé par le MTA.

III.2.4. Description de l'agent "MTA".

a) Déclaration de l'agent.

Le tableau 3.3. de l'annexe 2 donne la liste des composantes d'état et des actions qui sont sous la responsabilité de l'agent MTA, avec une brève description de celles-ci. Quant à la description des éléments importés, le lecteur peut se référer au tableau de l'agent exportant se trouvant également à l'annexe 2. Les définitions des types abstraits de données se trouvent à l'annexe 3.

La figure 3.7. représente la déclaration graphique de l'agent MTA en langage ALBERT.

Remarques:

Le MTA-messages est une population constituée d'un ensemble de messages, à chacun desquels est associé un ensemble de destinataires. Ceux-ci sont soit des agents UA, soit des agents MTA.

De la même manière, le MTA-reports est une population constituée d'un ensemble de rapports de livraison/non-livraison, à chacun desquels est associée un et un seul destinataire: un agent UA ou un agent MTA. Dans ce dernier cas, notons qu'il ne s'agit pas du destinataire final, mais bien d'un intermédiaire par lequel transite le message avant d'arriver au destinataire prévu.

b) Spécification des contraintes.

La figure 3.8. reprend la spécification des contraintes.

MTA AGENT CONSTRAINTS

BASIC CONSTRAINTS

Derived Components

Initial Conditions

MTA-messages[-] = UNDEF
MTA-reports[-] = UNDEF

LOCAL CONSTRAINTS

States Behaviour

*

MTA-messages[-]<> UNDEF ⇒ ◊ (MTA-messages[-]= UNDEF)

*

MTA-messages[-]<> UNDEF ∧ In(MTA-messages[-],ua)
⇒ In(Recip(Enveloppe(MTA-messages[-])),ua)

Figure 3.8.: Spécification des contraintes pour l'agent UA en ALBERT.

*

MTA-reports[r]<> UNDEF ⇒ ◊ (MTA-reports[r]= UNDEF)

*

In(ua,UAs-know) ⇒ □ (In(ua,UAs-know))

*

In(mta,MTAs-know) ⇒ □ (In(mta,MTAs-know))

Effects Of Actions

* EA1: Quand une soumission est perçue par le MTA, il stocke le message et il initialise la liste des destinataires à { }.
SUBMIT-Conf(ua,m,ok): MTA-messages[m] = { }

* EA2: Quand un message est délivré à un UA, il le retire de la liste des destinataires de ce message.
DELIVER-Ind(ua,m): Remove(MTA-messages[m],ua)

* EA3 Quand un message est délivré à un UA et que cet UA est le dernier destinataire auquel doit être délivré ce message, il supprime ce message de MTA-messages..
DELIVER-Ind(ua,m) with MTA-messages[m] = { }:
MTA-messages[m] = UNDEF

* EA4
NOTIFY-Ind(ua,r,-): MTA-reports = UNDEF

* EA5: Action générant un rapport de livraison/non-livraison
Report-generate(r,ua,m): MTA-reports[r] = ua

* EA6: Action générant la liste destinataires directes du message. Le destinataire est UA si celui-ci est local au MTA ou un MTA MTA adjacent si l'UA n'est pas local.
Message-routing(ua,m,dest) with ¬In(dest,MTA-messages[m]):
Add(MTA-messages[m],dest)

* EA7
Message-transfer(mta,mta',m): Remove(MTA-messages[m],ua)

* EA8
Message-transfer(mta,mta',m) with MTA-messages[m] = { }:
MTA-messages[m] = UNDEF

* EA9
mta.Message-transfer(mta,mta',m): MTA-messages[m] = { }

Causality

* C1: Quand le MTA perçoit une action de soumission de message provenant d'un UA, alors les opérations suivantes se dérouleront

ua.SUBMIT-Req(m) $\xrightarrow{\Diamond}$ SUBMIT-Conf(m,r,ua);
∀ ua' ∈ Recip(Enveloppe(m)):
[
Message-routing(m,ua,dest);Enveloppe-modify(m,m');
{Message-transfert(m',dest) with Is-of_MTA(dest) ⊕
{DELIVER-Ind(m',ua) ⊗
(Report-generate(Origin(Enveloppe(m')),r,Messid(m')) ⊕ DAC)} ⊕
(Message-lost(m') ⊗ Report-generate(r,Origin(Enveloppe(m')),m'))
}]

* C2: Si un rapport est généré, il devra être notifié à l'UA pour qui ce rapport est destiné.
Report-generate(m,ua) $\xrightarrow{\Diamond}$ NOTIFY-Conf(r,ua)

* C3
ua.LOGON-Req(mta,ap) $\xrightarrow{\Diamond}$ LOGON-Conf(r,ua)

* C4
ua.LOGOFF-Req(mta) $\xrightarrow{\Diamond}$ LOGOFF-Conf(ua)

Figure 3.8.: Spécification des contraintes pour l'agent UA en ALBERT (suite).

Capability

* Cap1: Un SuBMIT est accepté si le UA est connu du MTA (local au MTA)
  F(**SUBMIT-Conf(m,r,ua)** / ¬ In(ua,UAs-know))
* Cap2
  F(**DELIVER-Ind(m,ua)** / ¬In(ua,UAs-know) ∨ MTA-messages[m] = UNDEF ∨
  ¬In(MTA-messages[m],ua))
* Cap3
  F(**NOTIFY-Ind(r,ua)** / ¬ In(ua,UAs-know) ∨ ¬ MTA-reports[r] = UNDEF ∨ ¬In(MTA-reports[r],ua))
* Cap4
  F(**Report-generate(r,ua,m)** / ¬ In(ua,UAs-know) ∨ ¬ MTA-messages[m] = UNDEF ∨
  Origin(Enveloppe(m) <> ua)
* Cap5
  F(**LOGON-Conf(r,ua)** / ¬ In(ua,UAs-know))
* Cap6
  F(**LOGOFF-Conf(ua)** / ¬ In(ua,UAs-know))
* Cap7
  F(**Message-transfert(m,mta,-)** / ¬In(mta,MTAs-know) ∨ MTA-messages[m] = UNDEF ∨
  ¬In(MTA-messages[m],mta))

COOPERATION CONSTRAINTS

Action Perception

* AP1
  I(**ua.SUBMIT-Req(mta,m)** / ¬ In(ua,UAs-know) ∨ mta <> self)
* AP2
  I(**ua.LOGON-Req(mta,-)** / ¬ In(ua,UAs-know) ∨ mta <> self)
* AP3
  I(**ua.LOGOFF-Req(mta)** / ¬ In(ua,UAs-know) ∨ mta <> self)
* AP4
  I(**mta'.Message-transfert(m,mta',mta)** / ¬ In(mta,(MTAs-know) ∨ mta <> self ∨ MTA-[m] <> UNDEF)

State Perception

Action Information

* AI1
  XK(**SUBMIT-Conf(ua,m).ua'** / In(ua,UAs-know) ∧ ¬In(ua,MTA-messages[m]) ∧
  MTA-messages[m] = UNDEF ∧ ua = ua')
* AI2
  XK(**DELIVER-Ind(m,ua).ua** / In(ua,UAs-know) ∧ In(ua,MTA-messages[m]) ∧
  MTA-messages[m] <> UNDEF ∧ ua = ua')
*AI3
  XK(**NOTIFY-Ind(r,ua,-).ua'** / In(ua,UAs-know) ∧ In(ua,MTA-reports[m]) ∧
  MTA-reports[m] <> UNDEF ∧ ua = ua')
* AI4
  XK(**LOGON-Conf(ua).ua'** / In(ua,UAs-know) ∧ ua = ua')
* AI5
  XK(**LOGOFF-Conf(ua).ua'** / In(ua,UAs-know) ∧ ua = ua')
* AI6
  XK(**Message-transfert(m,mta,mta').mta''** / mta = self ∧ In(mta',MTAs-know) ∧
  In(mta',MTA-messages[m]) ∧ mta' = mta'')

State Information

Figure 3.8.: Spécification des contraintes pour l'agent UA en ALBERT (suite).

Remarques:

Décrivons, en langage ALBERT, le routage des messages dans le MTA d'une manière déclarative et simplifiée. Lorsqu'un message soumis au MTA a été accepté, l'action interne Message-routing est exécutée. Pour chaque destinataire, cette action recherche, à partir des informations contenues dans le MTA, quel sera le chemin à suivre par le message. Pour ce faire, elle vérifie si l'UA du destinataire est directement relié au MTA (UA inclu dans la population UAs-know). Si tel est le cas, la valeur de ce UA sera placée dans l'ensemble des destinataires du MTA-message pour ce message. Si tel n'est pas le cas, elle détermine vers quel MTA repris dans la population MTAs-know le message devra être guidé. La valeur de ce MTA choisi est placée dans l'ensemble des destinataires du MTA-message pour ce message. C'est la contrainte de causalité qui garanti les actions proprement dites de transfert de message entre MTA ou de livraison de celui-ci à l'UA. Notons que dans ce dernier cas, certaines conditions doivent être remplies: les instances Deliver-and-notify et Logon doivent avoir la valeur TRUE.

III.2.5. Description de l'agent "DS".

Déclaration de l'agent.

Le tableau 3.4. de l'annexe 2 donne la liste des composantes d'état et des actions qui sont sous la responsabilité de l'agent UA, avec une brève description de celles-ci. Quant à la description des éléments importés, le lecteur peut se référer au tableau se trouvant également à l'annexe 2. Les définitions des types abstraits de données se trouvent à l'annexe 3.

La figure 3.9. représente la déclaration graphique de l'agent DS en langage ALBERT.

Nous n'approfondirons pas les contraintes relatives au DS étant donné que la description des actions qui s'y déroulent ne sont pas d'un intêret capital dans le cadre de ce travail. Remarquons par ailleurs que ces actions ne sont que des réponses aux requêtes de l'agent UA

III.2.6. Description des buts du système.

Etant donné que nous disposions d'un système en place, notre démarche s'est en quelque sorte déroulée en sens inverse. En effet, la structure et les buts du système avaient déjà été pensés. D'autre part, la complexité du système est telle qu'il est difficile de faire ressortir formellement les buts avant de ne comprendre suffisament ses agents.

Voici exposés maintenant quelques uns des buts du système spécifié dans ce chapitre. Ceux-ci ont été choisis parmi les autres car ils concernent plus directement le transport de messages sécurisés dans le système. Ils seront repris et modifiés pour s'adapter aux nouveaux besoins relatifs aux services de sécurité de X.400-88.

Buts déduits de la spécification ALBERT de EAN:

* Un message stocké dans "Stored-messages" d'un UA et reçu par cet UA, a du être stocké dans le MTA local à cet UA.
us: USER,ua: UA, mta: MTA, mid: MESS-ID, m: MESSAGE,
us.UA-accessible = ua ∧ ua.Stored-messages[mid] = m ∧ In-out(ua.UA-Header[mid]) = 'in'
⇒ mta: mta.MTA-messages[m] = ua

* Un message stocké dans un MTA et devant être transféré ou délivré, a du être stocké soit dans un autre MTA, soit dans le Stored-messages" de l'UA expéditeur.

mta: MTA, dest: DEST, m: MESSAGE,
mta.MTA-messages[m] = dest ⇒
(♦ ∃ mta': mta'.MTA-messages[m] = mta) ∨
(♦ ∃ ua: UA,mid: MESS-ID,us: USER: ua.Stored-messages[mid] = m ∧
In-out(ua.UA-Header[mid] = 'out' ∧ us.UA-accessible = ua)

* Un rapport de livraison/non-livraison stocké dans "Stored-reports" d'un UA et se rapportant à un message soumis par cet UA, a du être stocké dans le MTA local à cet UA.

us: USER,ua: UA, mta: MTA, mid: MESS-ID, r: REPORT,
us.UA-accessible = ua ∧ ua.Stored-reports[mid] = r ∧ ua.Stored-messages[mid] m m ∧
In-out(UA-Header[mid]) = 'out' ∧ Report(Status(UA-Header(mid]))) = TRUE
⇒ ♦ mta.MTA-reports[r] = ua

III.3. Description générale de la soumission d'un message.

A la figure 3.10. est résumé la succession des actions que l'UA réalise et perçoit lorsqu'un utilisateur désire envoyer un message.

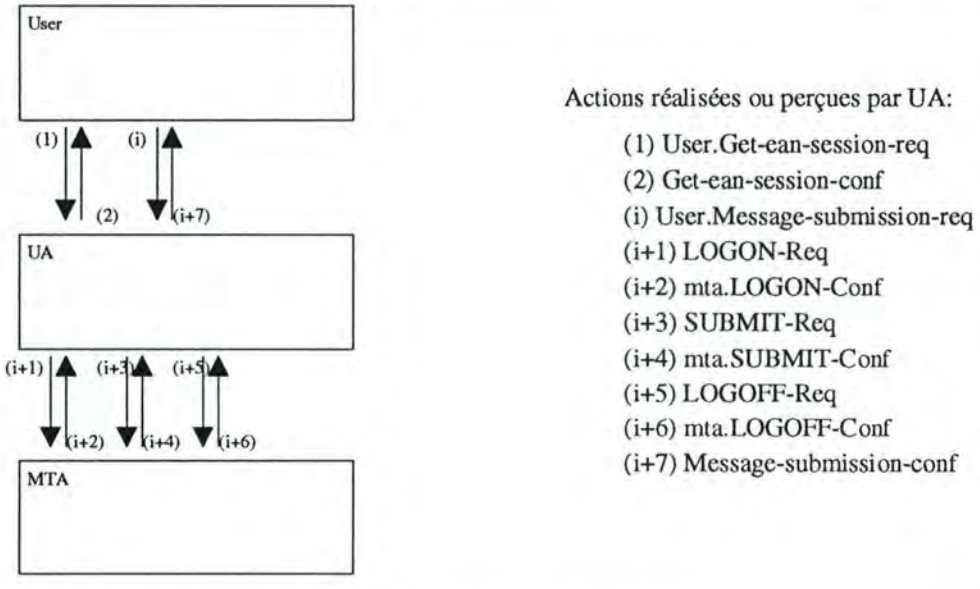

Figure 3.10.: Etapes possibles pour soumettre un message.

Une fois la connexion au logiciel réalisée et la demande de soumission du message au MTA de la part de l'utilisateur effectuée, une série d'étapes successives vont être réalisées:

-(i+1) **LOGON-Req**: demande de connexion de l'UA au MTA,

-(i+2) **LOGON-Conf**: confirmation à l'UA, par le MTA, de la demande de connexion,

-(i+3) **SUBMIT-Req**: soumission du message au MTA par l'UA,

-(i+4) **SUBMIT-Conf**:confirmation à l'UA, par le MTA, de la soumission du message,

-(i+5) **LOGOFF-Conf**: demande de déconnexion de l'UA au MTA,

-(i+6) **LOGOFF-Conf**: confirmation à l'UA, par le MTA, de la demande de déconnexion,

-(i+7) **Message-submission-conf**: confirmation à l'utilisateur de l'acceptation de la soumission du message.

Les actions(1), (i), (i+2), (i+4) et (i+6) sont perçues par l'UA alors que les actions (2), (i+1), (i+3), (i+5) et (i+7) sont celles qu'il exécute.

On remarquera la similitude que présente cette description de succession d'actions formulées en langage ALBERT avec le modèle en couches décrit classiquement dans X.400-84.

Nous ajouterons un nouvel "interlocuteur" dans ce double dialogue lorsque nous spécifierons les besoins en sécurité pour notre système (chapitre VI). Avant cela, nous introduirons des notions de cryptologie (chapitre IV) et proposerons un aperçu des services de sécurité prévus par X.400-88 (chapitreV) qui nous permettrons d'aborder respectivement les fonctions de bas niveau et celles de haut niveau du serveur de sécurité (chapitre VI).

Chapitre IV:

Eléments de cryptologie.

IV.1. Introduction.

Depuis des millénaires, la **cryptographie** est l'art de communiquer de façon confidentielle par l'entremise de voies de communications susceptibles d'espionnage. La **cryptanalyse** est l'art complémentaire consistant à décrypter de telles communications sans en être le destinataire légitime. Historiquement, la **cryptologie** ou **science du chiffre**, regroupant la cryptographie et la cryptanalyse, a été presqu'exclusivement l'apanage des militaires et des diplomates. Il est intéressant de constater que ce sont précisément les efforts accomplis pour décrypter les messages allemands qui furent le moteur du développement des premiers véritables ordinateurs. Depuis l'avènement de ceux-ci, et encore plus d'une société dans laquelle des masses d'informations personnelles, financières, commerciales et techniques sont conservées dans des banques de données informatiques et transférées à travers les réseaux d'un ordinateur à l'autre, la nécessité d'une cryptographie civile est devenue indispensable. David Khan, l'historien de la cryptographie par excellence, a écrit à ce sujet "La cryptographie, qui en 1945 était l'un des secrets les mieux gardés des nations, fait maintenant partie du domaine public" [DKa83].

IV.2. Principes de base.

Le but d'un **système cryptographique** (aussi appelé **cryptosystème**) est de **chiffrer** un **message intelligible** (aussi appelé **texte clair**) en un **texte chiffré** incompréhensible (aussi appelé **cryptogramme**). Le destinataire légitime doit pouvoir **déchiffrer** le cryptogramme et obtenir le texte clair. Cependant, un espion (aussi appelé **cryptanalyste**, décrypteur ou oreille indiscrète) ne doit pas être en mesure de décrypter le texte chiffré. Il faut donc bien distinguer déchiffrement (opération effectuée par le destinataire légitime) et décryptement (opération que l'espion tente d'effectuer).

Il existe plusieurs types de cryptosystèmes. Le classement suivant reprend les différents types de cryptosystèmes connus à ce jour:

-les cryptosystèmes à usage restreint,
-les cryptosystèmes à usage général:
    à clé secrète (aussi appelés **symétriques**),
    à clé publique (aussi appelés **asymétriques**).

Un système cryptographique est dit à **usage restreint** si sa sécurité repose sur la confidentialité des opérations de chiffrement et de déchiffrement. Le plus simple des systèmes historiques de ce genre est le procédé de substitution dit de Jules César. Il consiste simplement à remplacer chaque lettre du texte clair par celle qui la suit trois lettres plus loin dans l'alphabet (en revenant au début si nécessaire, c.-à-d. que x,y et z sont chiffrés par a,b et c respectivement). Les systèmes à usage restreint sont souvent conçus par des amateurs et sont presque toujours un jeu d'enfant pour les cryptanalystes professionnels.

Un système cryptographique est dit à **usage général** si sa sécurité ne repose pas sur le secret des opérations de chiffrement et de déchiffrement mais plutôt sur une information appelée la **clé**, laquelle est souvent relativement courte. Les personnes qui utilisent de tels systèmes doivent pouvoir facilement générer leurs propres clés sans avoir recours au concepteur du système, de sorte que celui-ci ne jouisse d'aucun avantage particulier s'il décide de passer dans le camp des cryptanalystes.

Un système cryptographique à usage général est dit à **clé secrète** si celle-ci doit faire l'objet d'une entende préalable entre ceux qui désirent utiliser le système pour communiquer. Actuellement, il est devenu quasiment impensable d'établir un réseau dans lequel chaque paire d'utilisateurs potentiels devrait partager une clé secrète établie à l'avance. Ainsi, W. Diffie et M. Hellman ont inventé un système cryptographique à usage général et à **clé publique**.

L'observation fondamentale qui sous-tend ce principe est que celui qui chiffre un message n'a nullement besoin d'être en mesure de le déchiffrer. Chaque utilisateur choisit une **clé personnelle** à partir de laquelle il déduit une paire d'algorithmes: l'algorithme public de chiffrement, qui est rendu disponible à tous, et l'algorithme privé de déchiffrement, qui est gardé secret par chaque utilisateur. Ceci permet à un parfait inconnu d'utiliser l'algorithme public pour chiffrer un message ; néanmoins, seul un utilisateur en règle est en mesure de le déchiffrer grâce à son algorithme privé. Cette méthode n'est valable que dans la mesure où l'effort requis pour reconstituer l'algorithme privé de déchiffrement à partir de l'algorithme public de chiffrement est démesuré.

IV.3. Les cryptosystèmes à clé secrète.

IV.3.1. Définitions.

Un cryptosystème à **clé secrète** est constitué d'un ensemble de clés K et, pour chaque clé $k \in K$, d'un ensemble M_k de textes clairs, d'un ensemble C_k de textes chiffrés, et d'une paire de fonctions:

$E_k: M_k \rightarrow C_k$ et

$D_k: C_k \rightarrow M_k$ telles que $D_k(E_k(m)) = m$ pour chaque texte clair $m \in M_k$. Le cryptosystème est **régulier** si M_k et C_k ne dépendent en fait pas de la clé k, c.-à-d. si des ensembles M et C existent tels que $M_k = M$ et $C_k = C$, pour tout $k \in K$.

De plus, il doit être facile d'obtenir des algorithmes efficaces pour calculer E_k et D_k à partir de toute clé k. Le cryptosystème est utilisé comme suit: si Alice et Bob ont quelque raison de penser qu'ils auront à communiquer secrètement un jour, ils doivent s'entendre au préalable sur une clé secrète $k \in K$; chaque fois qu'Alice souhaitera transmettre un $m \in M_k$ particulier à Bob, elle utilisera l'algorithme de chiffrement pour engendrer $c = E_k(m)$; elle enverra c à Bob sur un canal susceptible d'espionnage, et Bob utilisera l'algorithme D_k pour récupérer $m = D_k(c)$. La figure 4.1. donne une description classique de cette transmission confidentielle entre Alice et Bob.

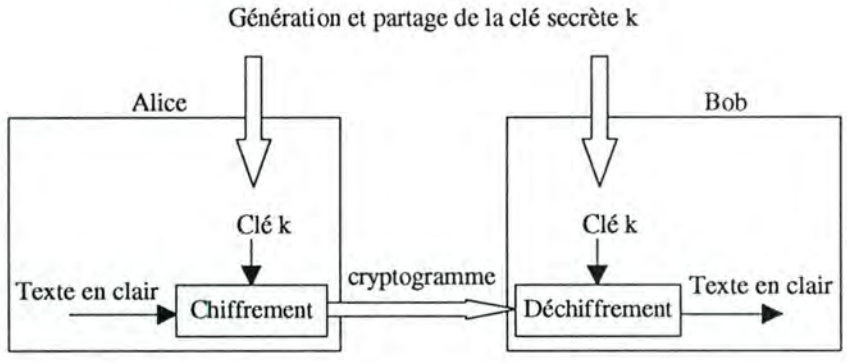

Figure 4.1.: Transmission confidentielle entre Alice et Bob.

La figure 4.2. en donne une description orientée agent suivant le langage ALBERT. Les populations Textes-clairs et Textes-chiffrés sont deux tableaux formés respectivement de la manière suivante: TAB[PERS → T-CL-PERS] et TAB[PERS → T-CH-PERS]. PERS, T-CL-PERS et T-CH-PERS sont des types abstraits représentant respectivement les agents "personne", les textes clairs associés à une personne et les textes chiffrés associés à une personne.

-Une opération particulière est associée au type T-CL-PERS:

Op $\mathbf{E_k}$: T-CL-PERS X PERS X KEY → T-CH-PERS.

Elle permet de chiffrer un message clair avec la clé k en vue d'obtenir un cryptogramme.

-Une autre opération particulière est associée au type T-CH-PERS:

Op $\mathbf{D_k}$: T-CH-PERS X PERS X KEY → T-CL-PERS.

Elle permet de déchiffrer ou de cryptanalyser un cryptogramme avec la clé k.

Voici exprimé les effets produits par les actions "Chiffrer", "Envoi-confidentiel", et "Réception-confidentielle", les capacités qu'un agent "personne" possède pour exécuter ces actions citées et les perceptions que cet agent recueille quant aux actions réalisées par un autre agent "personne":

Chiffrer(pers,k,i): tab-tch[i] = t-ch
with t-ch = E_k(tab-tcl[i] ∧
Textes-chiffrés[pers] = tab-tch ∧ Textes-clairs[pers] = tab-tcl

Déchiffrer(pers,k,i): tab-tcl[i] = t-cl
with t-cl = D_k(tab-tch[i] ∧
Textes-clairs[pers] = tab-tcl ∧ Textes-chiffrés[pers] = tab-tch

pers'.Envoi-confidentiel(pers',pers,t-ch,i): tab-tch[i] = t-ch with Textes-chiffrés[pers])=tab-tch

F(Envoi-confidentiel(pers,pers',t-ch,i) / tab-tch[i] <> t-ch with Textes-chiffrés[pers] = tab-tch)

I(pers'.Envoi-confidentiel(pers,pers",t-ch,i) / pers' <> pers ∨ self <> pers")

L'action Chiffrer signifie que l'agent chiffre son ième texte clair en un texte chiffré. L'action Envoi-confidentiel(pers,pers',i) signifie que l'agent Personne identifié par pers envoit un message chiffré à la personne identifié par pers'. Pour décrire le cas de la transmission confidentielle entre Alice et Bob, il faut poser pers = Alice et pers' = Bob.

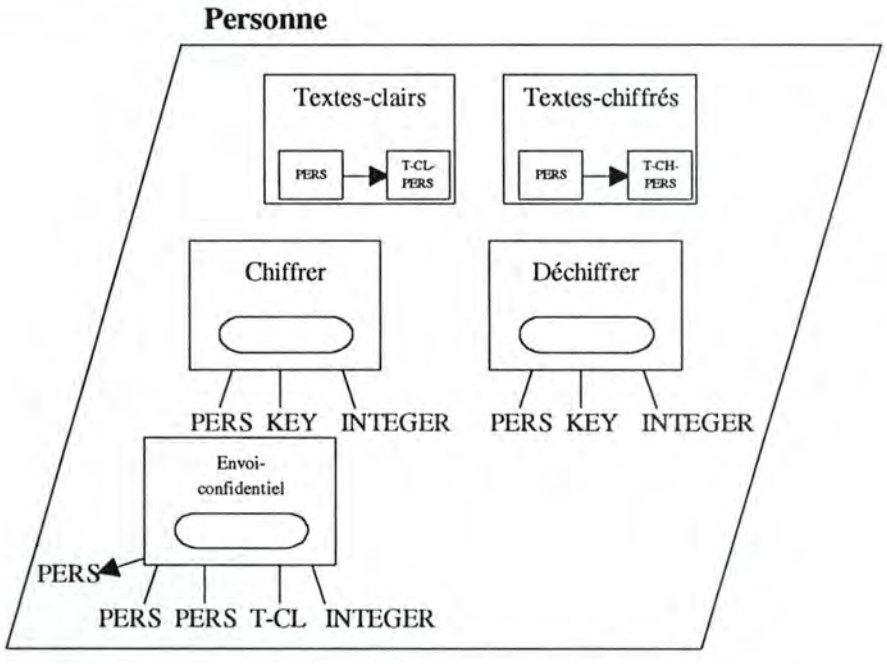

Figure 4.2.: Déclaration graphique en ALBERT de l'agent Personne.
permettant de décrire la transmission confidentielle entre Alice et Bob.

IV.3.2. Catégories d'attaques.

Ce que nous pourrions espérer d'un cryptosystème à clé secrète est que le cryptanalyste ne puisse déduire le texte clair m , ou encore la clé k suite à une interception du cryptogramme $c = E_k(m)$. Toutefois, même si un cryptosystème est à l'abri de cette menace, il peut se révéler vulnérable en d'autres circonstances: la cryptanalyse des systèmes à clé secrète distingue trois niveaux d'attaques classiques (un quatrième niveau sera à ajouter à ceux-ci dans le cas de cryptosystèmes à clé publique que nous décrirons plus loin). Ces niveaux d'attaques correspondent aux types d'informations dont le cryptanalyste dispose.

-L'**attaque à texte chiffré seulement** se produit lorsque le cryptanalyste dispose de $c_1 = E_k(m_1)$, $c_2 = E_k(m_2)$,..., $c_i = E_k(m_i)$, c.-à-d. des résultats du chiffrement de i messages inconnus avec une clé inconnue.

-L'**attaque à texte clair connu** se produit lorsque le cryptanalyste dispose de c_1, c_2,..., c_i comme ci-dessus, et qu'en plus il connaît les m_1, m_2,..., m_i correspondants.

-L'**attaque à texte clair choisi** se produit lorsque le cryptanalyste peut choisir des textes clairs m_1, m_2,..., m_i arbitraires et obtenir les $c_1 = E_k(m_1)$, $c_2 = E_k(m_2)$,..., $c_i = E_k(m_i)$ correspondants. Nous pouvons distinguer deux types d'attaques à texte clair choisi:

-L'**attaque statique**: le cryptanalyste doit choisir à l'avance m_1, m_2,..., m_i et il reçoit ensuite les textes chiffrés correspondants.
-**L'attaque dynamique**: le cryptanalyste choisit m_1 et il reçoit le c_1 correspondant avant d'avoir à choisir m_2; d'une façon générale m_j peut dépendre d'un ou plusieurs cryptogrammes c_i avec $i \leq j$.

IV.3.3. Modes d'opération.

La plupart des cryptosystèmes utilisés en pratique sont réguliers, et de plus M et C sont souvent des ensembles finis. Un de ces ensembles pourrait par exemple contenir toutes les suites de huit caractères. Dans ce cas, il est possible que le message m soit trop long pour être chiffré directement. Si cela se produit, m doit être découpé en tranches et E_k utilisé plusieurs fois.

Quatre grands modes de chiffrement du message, appelés **modes d'opération**, peuvent être utilisés en fonction de la manière dont on tient compte des messages découpés. Décrivons deux de ces modes:

-le mode d'opération "**carnet de codage électronique**" (**ECB** pour **Electronic Code Book**) permet de découper le message en tranches et de chiffrer chacune d'elles indépendamment en utilisant la même clé. La faiblesse la plus évidente est que deux tranches identiques dans le texte chiffré indiquent clairement au cyptanalyste que les tranches correspondantes du texte clair sont elles-mêmes identiques. Une telle information offre un point d'attaque sérieux pour retracer le texte clair. La situation est pire encore pour les problèmes d'authentification (cf. IV.3.1.) puisqu'elle donne prise à la technique du couper/coller.

-le mode d'opération "**chiffrement avec chaînage des blocs**" (**CBC** pour **Cipher Block Chaining**) permet de découper le texte clair en m blocs tel que $m = m_1 m_2 ... m_n$ et, pour chaque $1 \leq i \leq n$, d'obtenir les blocs chiffrés c_i correspondants aux blocs clairs m_i de la manière suivante: $c_i = E_k(m_i \oplus c_{i-1})$, où "$\oplus$" représente le ou-exclusif bit à bit et c_0 un bloc initial (non nécessairement secret). Le texte chiffré résultant est $c = c_1 c_2 ... c_n$. Les valeurs de k et c_0 étant données, le déchiffrement s'effectue en calculant $m_i = c_{i-1} \oplus D_k(c_i)$. En mode CBC, une erreur intervenant dans un bloc affecte le déchiffrement de deux blocs (le bloc erroné et le suivant).

Les deux autres modes d'opération sont le "**chiffrement par rétroaction**" (**CFB** pour **Cipher FeedBack**) et le "**chiffrement par rétroaction de la sortie**" (**OFB** pour **Output FeedBack**). Ils permettent la limitation de la propagation des erreurs. Une description de ces deux modes est donnée dans [RH93].

IV.3.4. Le Standard de Chiffrement de Données.

IV.3.4.1. Confusion et diffusion.

Le but de la diffusion est de dissiper la redondance de la langue naturelle utilisée dans le texte clair en l'étalant à travers tout le texte chiffré.

Le but de la confusion est de rendre la relation entre la clé et le texte chiffré aussi complexe que possible, avec pour résultat que le cryptanalyste n'obtiendra que très peu d'information utile sur la clé à partir d'une analyse statistique du texte chiffré. La confusion est généralement réalisée au moyen de la technique de substitution. Cette technique consiste à

remplacer les caractères d'un texte clair par d'autres caractères selon une loi bien définie (procédé de Jules César).

IV.3.4.2. Description du Standard de Chiffrement de Données.

Le Standard de Chiffrement de Données (DES ou Data Encryption Standard) est un célèbre système à clé secrète mis de l'avant en 1977 par le Bureau National américain des Standards (NBS). Il fut conçu pour être utilisé pendant dix à quinze ans pour "fins internes du Gouvernement Fédéral (américain) pour la protection cryptographique des données informatiques de nature sensible mais non secrète". Bien que ce laps de temps se soit écoulé, le DES demeure très utilisé en pratique, en particulier dans le domaine commerciale et bancaire. Son principal avantage est d'atteindre de très **grandes vitesses** de chiffrement et de déchiffrement. Nous n'entrerons pas dans les détails de fonctionnement de l'algorithme DES. De très bonnes descriptions se trouvent dans [RH93]. Contentons-nous de dire qu'il chiffre un bloc de 64 bits à l'aide d'une clé secrète de 56 bits (en fait 64 bits qui incluent 8 bits de parité). L'algorithme DES transforme la clé en seize clés partielles de 48 bits au moyen d'**un plan de génération des clés** qui réutilise chacun des bits de la clé plusieurs fois. Après une permutation initiale fixe de **diffusion**, le bloc de 64 bits de texte clair fait l'objet de seize rondes suivies de l'inverse de la permutation initiale de diffusion. Chaque ronde effectue une étape de **confusion** (à l'aide des tables-S et de la clé partielle correspondante), suivie d'une étape de **diffusion**. Il est remarquable que l'étape de confusion ne dépende pas de la clé; la sécurité du système est augmentée en combinant confusion et diffusion même si l'une des deux transformations est fixe et publiquement connue. L'algorithme DES est conçu de telle sorte que le déchiffrement s'effectue exactement de la même manière que le chiffrement, sauf que l'ordre des clés partielles est inversé dans le plan de génération des clés. Ceci est pratique puisque le même dispositif peut servir à la fois pour le chiffrement et le déchiffrement.

IV.3.4.3 Performance du Standard de Chiffrement de Données.

Comme nous l'avons déjà signalé, le DES permet d'atteindre de très grandes vitesses de chiffrement et de déchiffrement lorsqu'il est réalisé sur du matériel spécialisé. La vitesse maximale de chiffrement connue à ce jour pour une puce est de 90 mégabits par seconde. [Mac81] décrit une puce capable de chiffrer à la vitesse 14 Mb/s. De telles vitesses sont suffisamment élevées pour pouvoir chiffrer ou déchiffrer à la volée, lorsque les données sont lues ou écrites sur disque et, sont satisfaisantes pour la plupart des besoins de télécommunication. Il est même possible d'atteindre des vitesses acceptables au moyen de logiciels: jusqu'à 650 kb/s sur un IBM PS/2 modèle 80 [GVBP90].

IV.4. Les cryptosystèmes à clé publique.

IV.4.1. Fonctions à sens unique et fonctions à brèche secrète.

Les notions de fonction à sens unique et de fonction à brèche secrète sont au coeur de la cryptographie à clé publique.

Considérons deux ensembles arbitraires X et Y, et une fonction $f:X \rightarrow Y$. Soit f[X] l'image de f. La fonction f est dite **à sens unique** s'il est facile de calculer f(x) pour tout $x \in X$, alors qu'il est difficile de trouver, pour la plupart des $y \in f[X]$ un $x \in X$ tel que $f(x) = y$. Attirons l'attention sur le fait que cette notion ne doit pas être confondue avec les fonctions qui, au sens mathématique du terme, n'ont pas d'inverse.

La certitude de l'existence de fonctions à sens unique suffirait à démontrer que P <> NP[GJ79]. Malheureusement, l'état actuel de nos connaissances en théorie de la complexité du calcul ne nous permet pas d'en prouver leur existence. Il nous reste néanmoins possible de calculer efficacement des fonctions candidates pour lesquelles aucun algorithme efficace capable de les inverser n'est connu à ce jour.

Un exemple simple de fonction potentiellement à sens unique est la **multiplication des entiers**. Multiplier deux entiers de cent chiffres est facile, même avec un ordinateur modeste. Par contre, le plus puissant des ordinateurs disponibles, équipé du meilleur algorithme connu, serait incapable de factoriser, avant la fin prévue de l'Univers, le nombre ainsi produit. Dans [PST88], les auteurs décrivent une architecture pipe-line qui serait capable de factoriser ce nombre en un an de calcul, mais on estime le coût de construction de cette machine à environ cent milliards de dollars!!!

Evidemment, étant donné que même le destinataire légitime ne serait pas en mesure de récupérer le texte clair, ces fonctions à sens unique ne sont pas d'application immédiate en cryptographie. Par contre, les fonctions à brèche secrète se révèlent plus fonctionnelles en cryptographie à clé publique.

Une fonction f: $X \rightarrow Y$ est dite **à brèche secrète** si elle peut être calculée efficacement tant dans le sens direct que dans le sens inverse. Pour obtenir l'algorithme efficace permettant de calculer la fonction dans le sens inverse, la détention d'un secret, appelé brèche secrète, est indispensable à l'utilisateur.

IV.4.2 Description générale du cryptosystème à clé publique.

Très semblable au cryptosystème à clé secrète, le cryptosystème à **clé publique** est constitué d'un ensemble de clés K et, pour chaque clé $k \in K$, d'un ensemble M_k de textes clairs, d'un ensemble C_k de textes chiffrés, et d'une paire de fonctions:

$E_k: M_k \rightarrow C_k$ et

$D_k: C_k \rightarrow M_k$ telles que $D_k(E_k(m)) = m$ pour chaque texte clair $m \in M_k$.

A nouveau, il doit être facile d'obtenir des algorithmes efficaces pour calculer E_k et D_k à partir de toute clé k. Nous appellerons les algorithmes ainsi obtenus des **algorithmes naturels**. La différence avec les systèmes à clé secrète est que E_k doit être une fonction à brèche secrète. Il est impératif qu'on ne puisse déduire, à partir de l'algorithme naturel de calcul E_k, un quelconque algorithme efficace pour calculer D_k. Pour ce faire, la clé elle-même ne peut être déduite à partir de l'algorithme E_k. C'est ainsi que la fonction qui associe à k l'algorithme E_k devra être à sens unique et la clé k n'apparaîtra pas en clair dans l'algorithme naturel de déchiffrement.

Un système cryptographique à clé publique est utilisé comme suit. Une fois pour toute, l'utilisateur choisit au hasard une clé personnelle $k \in K$. Il l'utilise pour obtenir les deux algorithmes naturels E_k et D_k. Son algorithme de chiffrement est rendu publiquement disponible alors que son algorithme de déchiffrement est gardé secret. Lorsqu'un message lui est envoyé, l'expéditeur l'aura préalablement chiffré grâce à l'algorithme public du destinataire, trouvé aisément dans le répertoire et, c'est grâce à sa brèche secrète que seul le destinataire légitime pourra déchiffrer son message.

Les systèmes à clé publique n'existent que si les fonctions à sens unique et à brèche secrète existent. En fait, les fonctions de chiffrement doivent être à brèche secrète et le procédé qui livre l'algorithme naturel de chiffrement à partir de la clé doit être à sens unique. Il en résulte que dans l'état actuel de nos connaissances nous sommes incapables de démontrer l'existence des systèmes cryptographiques à clé publique.

Contrairement aux systèmes à clé secrète, remarquons que si Alice chiffre un message m destiné à Bob, qu'elle garde le texte chiffré c et perde le texte clair (en oubliant intégralement son contenu), elle se retrouvera dans la même situation qu'un cryptanalyste devant déterminer m à partir de c.

IV.4.3. Catégories d'attaques.

Une différence avec les systèmes à clé secrète est que si une oreille indiscrète intercepte un texte chiffré, elle peut utiliser l'algorithme public de chiffrement pour vérifier chaque conjecture spécifique qu'elle peut faire au sujet du texte clair. La capacité pour le cryptanalyste de produire le texte chiffré correspondant à tout texte clair de son choix efface les distinctions entre les divers niveaux d'attaques classiques dont nous avons parlé au sujet des systèmes à clé secrète. Cependant, l'attaque suivante est plus puissante:

-l'attaque à texte chiffré choisi: le cryptanalyste choisit des textes chiffrés c_1, c_2,..., c_i arbitraires et reçoit les $m_1 = D_k(c_1)$, $m_2 = D_k(c_2)$,..., $m_i = D_k(c_i)$ correspondants.

IV.4.4. Le système RSA.

IV.4.4.1. Description générale du système RSA.

Le tout premier système à clé publique à avoir été proposé dans la littérature scientifique fut celui de R. Rivest, A. Shamir et L. Adleman. [RSA78]. Il est maintenant connu sous le nom de système RSA ou MIT. Il s'appuie sur l'espoir que l'exponentiation modulaire avec un exposant et un module fixés est une fonction à brèche secrète.

Soient p et q deux grands nombres premiers distincts. Soit n = pq, et soit e un entier quelconque premier à (p-1)(q-1). Chaque triplet k = <p,q,e> est une **clé personnelle** du système RSA. L'ensemble des messages M_k et l'ensemble des textes chiffrés C_k coïncident tous les deux avec Z_n, l'ensemble des entiers non négatifs inférieurs à n. Si le message est trop long pour appartenir à Z_n, il devra être découpé en tranches et chiffré selon le mode de chiffrement par chaînage des blocs (CBC) décrit au paragraphe IV.3.3. La **fonction de chiffrement** correspondant à la clé k est E_k: $M_k \rightarrow C_k$ définie par $\mathbf{E_k(m) = m^e \bmod n}$. Pour être en mesure de la calculer, il suffira d'avoir enregistré n et e dans le répertoire public. <n, e> constitue donc la clé public. Celle-ci se déduit facilement de la clé personnelle <p,q,e>.

Nous avons vu que E_k est un candidat au titre de fonction à brèche secrète puisque, bien qu'existant , l'algorithme efficace permettant de calculer D_k, son inverse, ne peut être obtenu à partir des seules données n et e de l'algorithme naturel E_k. Par contre, il peut être obtenu facilement à partir des facteurs p et q de n. Pour ce faire, il suffit d'utiliser une extension de l'algorithme d'Euclide pour le calcul du plus grand commun diviseur afin de trouver un entier **d** tel que $\mathbf{ed \bmod \varphi = 1}$, où $\varphi = (p-1)(q-1)$. En vertu du théorème d'Euler, $m^{ed} \bmod n = m$ pour chaque entier m tel que $0 \leq m < n$, c.-à-d. pour tout $m \in M_k$. La **fonction de déchiffrement** est donc D_k: $C_k \rightarrow M_k$ définie par $\mathbf{D_k(c) = e^d \bmod n}$.

Pour résumer, chaque utilisateur du système RSA choisit au hasard une fois pour toute des entiers appropriés p, q et e, et calcule d à partir de ceux-ci. Il introduit sa clé publique <n,e> dans le répertoire des utilisateurs, et il garde d secret. Ce système permet à quiconque désire envoyer un message de le chiffrer à l'aide de la clé publique du destinataire que seul celui-ci sera capable de déchiffrer grâce à sa clé personnelle.

IV.4.4.2. Performance du système RSA.

Malgré tous les avantages que le système RSA présente par rapport aux systèmes à clé secrète, on notera qu'il est nettement plus lent que le DES. Rappelons que les implémentations du DES en matériel peuvent atteindre 90 mégabits/s. L'implémentation connue la plus rapide peut déchiffrer jusqu'à 225 kbits/s si le module est de longueur 508 bits [SBV90]. De plus, le déchiffrement est toujours plus rapide que le chiffrement puisqu'il existe un algorithme plus performant, algorithme utilisant p et q, connu uniquement au moment du déchiffrement. Il va s'en dire que les réalisations en logiciel sont plus lentes. L'implémentation logiciel du RSA la plus rapide avoisine les 5 kilobits/s pour le chiffrement et 12 kilobits/s pour le déchiffrement. Une approche qui permet d'accélérer grandement le chiffrement RSA (mais pas le déchiffrement) consiste à utiliser systématiquement un petit exposant public, par exemple e = 3. Il faut toutefois être conscient que le système devient alors vulnérable. Cette technique est fort utilisée pour les protocoles d'identification où les données chiffrées sont des défis pour l'utilisateur, permettant ainsi de l'identifier s'il arrive à surmonter ces défis. Une cryptanalyse avec e = 3 peut prendre quelques heures plutôt que quelques milliards d'années! Une période de quelques heures est cependant déjà une éternité pour une identification informatique.

IV.5. Applications.

Bien qu'historiquement la raison d'être de la cryptographie ait été l'établissement de communications confidentielles sur les voies de communications susceptibles d'espionnage, ce n'est certes pas son seul intérêt. D'autres applications comme l'authentification, la signature numérique ou l'identification utilisent ces techniques.

IV.5.1. L'authentification de document.

Jusqu'à maintenant, nous n'avons envisagé que le cas d'un cryptanalyste passif, c.-à-d. de quelqu'un dont le but est simplement d'espionner le canal de communication. Un cryptanalyste actif ne se contente pas d'écouter le canal de communication, il peut également modifier des messages légitimes en transit ou injecter des messages avec l'espoir de faire croire au destinataire qu'ils proviennent de quelqu'un d'autre. Le but d'un mécanisme d'authentification est de déceler la présence d'un cryptanalyste actif en vérifiant l'authenticité et l'intégrité du message transmis. Voici quelques exemples de protocoles d'authentification de documents parmi les plus utilisés:

-Authentification sans élément secret.

Cette technique se base sur les codes détecteurs. Elle permet d'élaborer un système assez simple où il suffit simplement de calculer un paramètre caractéristique du texte (code détecteur par exemple), de l'envoyer d'une manière sûre, et d'envoyer le message sur le canal non sécurisé. A la réception du message, on recalcule ce paramètre et on le compare avec celui reçu et dont on est sur qu'il n'a pas été modifié. Si les deux paramètres sont égaux alors le message est authentique.

-Authentification par chiffrement et code détecteur.

Cette méthode est employée lorsque l'on désire envoyer un message entièrement chiffré. Elle consiste à calculer un paramètre caractéristique du texte clair M et à envoyer l'ensemble formé par le message et le paramètre sous forme chiffrée . Ce paramètre est souvent appelé **Manipulation Detection Code** (**MDC**). Dès que le

destinataire aura reçu le cryptogramme, il le déchiffrera afin de récupérer le code détecteur et le message en clair. Ensuite, il calculera un nouveau code pour le message et le comparera avec le code qu'il venait d'obtenir par déchiffrement. S'ils sont différents, le texte clair reçu a été manipulé et n'est donc pas authentique. L'avantage de la technique du MDC est qu'il est facile d'avoir une réponse (vrai ou faux) sur l'authenticité du message: il suffit de calculer le code du message reçu et de le comparer au code reçu. Par contre, si on envoie uniquement un message chiffré sans MDC, la détermination de l'authenticité du message est plus délicate. En effet, l'authenticité de celui-ci sera vérifiée si le texte résultant du déchiffrement du message reçu a un sens.

-Authentification avec fonction cryptographique à sens unique.

Cette méthode peut être utilisée lorsque l'on ne désire pas envoyer un message chiffré. Elle consiste à calculer un paramètre caractéristique du texte clair et à l'envoyer avec le texte en clair. Ce paramètre est souvent appelé **Manipulation Authentication Code** (**MAC**). La fonction qui permet de calculer ce paramètre doit être une **fonction à sens unique** (définition au paragraphe IV.4.1.), paramètrée par une clé propre à l'expéditeur et au destinataire. Le MAC est donc un condensé cryptographique du message en clair. A la réception du message par le destinataire, celui ci recalcule le MAC sur le message reçu avec la même clé et compare celui-ci avec le MAC accompagnant le message reçu. S'ils sont différents le message aura été manipulé durant son transport. Le calcul du MAC utilise les algorithmes traditionnels de la cryptographie: c'est ainsi que le DES peut être adapté à cette fin.

IV.5.2. La signature digitale.

Bien que le mécanisme d'authentification permette au destinataire du message de s'assurer que le message reçu provienne bien de l'expéditeur, il ne lui permet pas de convaincre autrui que tel est effectivement le cas. Les mécanismes d'authentification ne sont donc utiles qu'à deux individus qui se font mutuellement confiance. Ils ne permettent pas de régler des disputes entre eux. L'utilisation des fonctions à brèche secrète a rendu possible la notion de signature numérique. Voici quelques exemples de protocoles de signature digitale les plus utilisés:

-Signature digitale par chiffrement asymétrique.

Souvenons-nous de ce que nous avons vu précédemment (cf. IV.4.3.). Une fois pour toute, l'utilisateur choisit au hasard une clé personnelle $k \in K$. Il l'utilise pour obtenir les deux algorithmes naturels E_k et D_k. Son algorithme de chiffrement est rendu publiquement disponible alors que son algorithme de déchiffrement est gardé secret. Pour signer un message, le rôle des deux algorithmes naturels est inversé. L'algorithme public de chiffrement E_k devient l'algorithme public de déchiffrement $\mathbf{V_k}$, lequel permet de vérifier la signature. L'algorithme secret D_k de déchiffrement, quant à lui, devient l'algorithme secret de chiffrement $\mathbf{S_k}$, lequel permet de générer la signature.

-Signature digitale par chiffrement asymétrique et code détecteur.

Tout comme nous l'avons fait pour l'authentification, nous pouvons ajouter au protocole précédent un code détecteur (MDC) permettant de vérifier plus facilement la cohérence du message. Le bloc de données, contenant le message et le code détecteur,

est signé (chiffré) à l'aide de l'algorithme secret de l'expéditeur, permettant ainsi à toute les personnes percevant ce bloc de données de le déchiffrer à l'aide de l'algorithme public de vérification de signature, de recalculer le code détecteur pour le message reçu et de vérifier si ce code détecteur est semblable au code détecteur reçu. Si les deux codes sont égaux, alors l'expéditeur a réellement envoyé le message sous cette forme.

-Signature digitale par compression et chiffrement asymétrique.

Les techniques précédentes ont toutes deux une caractéristique commune: le bloc de données à signer est chiffré. Il est cependant possible de ne chiffrer qu'une partie de ce bloc. Cette partie est en fait une compression du message obtenue par l'utilisation d'une fonction de hachage. Celle-ci doit être à sens unique construite de manière telle qu'elle évite à deux messages différents d'avoir le même résultat (Collision Free Hash Function [MRW89]).

IV.5.3. L'identification.

Le problème de l'identification est, à l'individu ce que la signature numérique et l'authentification sont à un document: comment un individu peut-il prouver son identité? Les applications de l'identification sont nombreuses. Par exemple, chaque fois qu'un utilisateur cherche à se brancher sur un ordinateur ou à accéder à une banque de données, qu'un client veut se servir d'un guichet automatique ou que le guichet désire accéder aux données de l'ordinateur central de la banque, comment s'assurer que l'accès ne se fasse pas sous une autre identité? Les protocoles d'identification se classent en trois grandes catégories:

-Identification par mot de passe.

C'est la méthode classique par laquelle un utilisateur transmet un mot de passe secret après avoir décliné son identité. Celui-ci est comparé à une liste conservée dans le système. Une amélioration consiste à stocker le mot de passe chiffré par une fonction à sens unique. Le mot de passe n'est dès lors plus stocké en clair dans le système. Dès que le système reçoit le mot de passe (en clair) d'un utilisateur, il applique la fonction à sens unique sur le mot de passe et obtient la version chiffrée. Il les compare alors pour vérifier l'identité de l'utilisateur.

-Identification par systèmes interactifs.

Ces systèmes font varier l'information transmise sur le réseau à chaque connexion. En effet, après réception de l'identité, le serveur envoie au correspondant une information aléatoire (une question). Ce dernier prouve son identité en répondant par cette question chiffrée à l'aide de sa clé secrète (équivalent de son mot de passe). Le serveur doit également disposer de cette clé afin de pouvoir effectuer le déchiffrement et la vérification de la réponse.

-Identification par transfert nul d'information.

Ces méthodes, aussi appelées protocoles "zero-knowledge", permettent d'éviter de placer la moindre information secrète dans le système mais cela au prix d'un plus grand nombre d'échanges d'informations. Après le transfert de l'identité, la vérification se fait en trois phases:

-le correspondant transmet une information aléatoire qui l'engage pour la suite du dialogue (commitment),
-le système pose ensuite une question aléatoire,
-la réponse dépend alors des deux messages précédents et d'une information secrète (mot de passe) attestant de l'identité.

En général, après un tel échange, il est possible de tricher dans un cas sur deux.(c.-à-d. de répondre correctement sans posséder l'information secrète). Le protocole est donc répété un certain nombre de fois afin d'atteindre le niveau de sécurité souhaité. (20 itérations laissent moins d'une chance sur un million à un faussaire). Le nom "apport nul de connaissance" provient du fait que la réponse n'apporte aucune information permettant de retrouver le secret. Un exemple de ce protocole est celui décrit par Fiat et Shamir dans [FS87].

Chapitre V:

La sécurité dans X.400-88.

De par leur nature distribuée, les réseaux de télécommunications et leurs applications (le MHS X.400 en particulier) sont sujets à des actions mal intentionnées et illégales. Il est donc nécessaire d'instaurer un système de défense. Dans X.400-84, aucune protection n'est fournie aux utilisateurs du MHS X.400. Nous allons décrire dans ce chapitre un modèle abstrait de sécurité introduit dans X.400-88 et qui utilise les mécanismes de chiffrement et de signature vus au chapitre précédent (cryptosystèmes symétriques et asymétriques). Les mécanismes de sécurité décrits dans la série de recommandations X.400 sont en grande partie basés sur les techniques cryptographiques à clé publique. Les services de sécurité du MHS permettent néanmoins la flexibilité dans le choix des algorithmes. Nous illustrerons également dans ce chapitre les attaques potentielles prévues par l'avis X.400-88, ainsi que leurs solutions cryptographiques (également décrites dans X.400-88).

V.1. Classification des attaques pouvant survenir au sein du MHS.

Trois classes d'attaques pouvant survenir au sein de MHS sont décrites par l'avis X.400-88.

V.1.1. Les attaques concernant l'accès au MHS.

La première menace au système est l'accès au MHS par des utilisateurs **non valides**. La minimisation de la probabilité de ces attaques est un élément essentiel pour la sécurité du MHS: elle réduit considérablement la probabilité des autres types d'attaques. En général, le contrôle d'accès au MHS dépend du système d'opération: pour accéder au MTS, l'utilisateur doit accéder au UA, programme s'exécutant sur une machine quelconque, laquelle peut se trouver au sein d'un réseau local.

V.1.2. Les attaques indiscrètes ("espion").

Ce type d'attaques peut être réalisé de deux manière différentes:

-lorsqu'une personne capte volontairement ou non un message qui ne lui était pas destiné, on parlera d'**indiscrétion passive,**
-lorsqu'un imposteur interrompt, modifie ou stimule une communication, on parlera d'**indiscrétion active**. X.400-88 distincte quatre types d'attaques entrant dans cette catégorie::

- **mascarade**: la mascarade se produit lorsqu'une entité particulière (UAE, MTAE) réussit à se faire passer pour une autre afin de soumettre, de transférer ou de délivrer des messages sous une fausse identité, ou encore, afin de consulter (MS) ou de se faire livrer (UA) des messages qui ne lui sont pas adressés . Des protocoles d'authentification d'acteur ou d'origine de message utilisant les méthodes de la cryptographie permettent de contrôler les tentatives de mascarades.

- **modification du message**: un message peut avoir été modifié d'une façon telle que la modification ne puisse être détectée. Cette modification peut concerner n'importe quelle partie du message: libellé, contenu, O/R name du destinataire, O/R name de l'expéditeur,...Bien qu'il ne soit pas possible d'éviter la modification de messages avec les services de sécurité du MHS, celle-ci peut néanmoins être détectée et l'effet de l'attaque éliminé par l'utilisation de méthodes cryptographiques et de signatures.

- **modification du séquençage de messages**: n'importe quelle partie d'un message peut être répétée, réordonnée ou décalée temporellement. Comme dans le cas précédent, il n'est pas possible d'éviter cette modification du séquençage des messages avec les services de sécurité du MHS; par contre, elle peut être détectée et l'effet de l'attaque éliminé par l'utilisation de méthodes cryptographiques et de signatures.

- **fuite d'information**: il est possible d'acquérir un message en l'interceptant lors de son transfert vers un autre MTA, lors de sa soumission ou lors de sa livraison; ou bien encore en accédant à l'endroit où il est stocké (dans une entité quelconque du MHS). D'autres informations, concernant le trafic des messages, l'identité des personnes utilisant le service de messagerie électronique,... peuvent être acquises de cette manière. Dans certains cas, l'anonymat d'un MTS-user, la confidentialité des messages et la confidentialité du trafic des messages sont essentiels. Pour réaliser ces services on utilise des méthodes de chiffrement.

V.1.3. Les attaques impliquant les acteurs d'un message (entre un expéditeur et ses destinataires).

Ces attaques mettent en jeu les participants réels du transfert de messages. X.400-88 distincte deux types d'attaques entrant dans cette catégorie:

- **réfutation**: on parle de réfutation lorsque le MTS-user ou le MTS nie avoir soumis un message, avoir reçu un message ou nie être l'expéditeur d'un message.

- **violation du niveau de sécurité**: si un domaine de gestion au sein du MHS emploie un niveau de sécurité parmi "public", "personal", "private",... alors tout envoi ou toute réception de messages ne répondant pas à ce niveau de sécurité doit être interdit.

V.2. Les services de sécurité du MHS.

Nous allons décrire brièvement les services de sécurité introduits dans X.400-88. Ces services protègent le MHS X.400 contre les nombreuses attaques pouvant survenir, comme par exemple l'interception et la modification d'un message par une personne "indiscrète",....

Les services de sécurité sont représentés par sept catégories:

1°) les services d'authentification de l'origine,
2°) les services de gestion d'accès sécurisé,
3°) les services de confidentialité des données,
4°) les services d'intégrité des données,
5°) les services de non-réfutation,
6°) les services de gestion de la sécurité,
7°) les services de sécurité concernant la catégorisation d'un message.

V.2.1. Définition des différents services de sécurité.

Le tableau 5.1. donne une brève définition des services de sécurité disponibles dans X.400-88. Afin de permettre au lecteur de les retrouver plus facilement dans la série de recommandations X.400-88, nous utiliserons les termes anglais.

Services de sécurité	**définition**
Services d'authentification de l'origine:	
Message Origin Authentification:	permet à l'UA destinataire, ou à un MTA à travers lequel passe le message d'authentifier l'identité de l'expéditeur.
Report Origin Authentification:	permet à l'UA expéditeur, ou à un MTA à travers lequel passe le rapport, d'authentifier l'origine d'un rapport de livraison/non-livraison.
Probe Origin Authenfication:	permet à un MTA à travers lequel passe la sonde (probe) d'authentifier l'origine de celle-ci.
Proof of Submission:	permet à l'expéditeur d'avoir la preuve que le message a été soumis au MTS.
Proof of Delivery:	permet à l'expéditeur d'avoir la preuve que le message a été délivré.
Services de gestion d'accès sécurisés:	
Peer Entity Authentification:	permet à chacun des interlocuteurs de s'assurer de l'identité de son correspondant. Elle évite de la sorte qu'une connexion (UA/MTA,MTA/MTA,MTA/UA) puisse être récupérée par un imposteur.
Security Context:	permet la mise en oeuvre du contexte de sécurité, la limitation du passage des messages entre composants suivant que le contexte de sécurité soit respecté ou non.
Services de confidentialité des données:	
Content Confidentiality	permet d'éviter la divulgation du contenu d'un message à un destinataire illégitime (y compris les MTAs).
Message Flow Confidentiality:	permet à l'expéditeur de dissimuler le flux de messages à travers le MHS.

Services de sécurité	définition
Services d'intégrité des données:	
Content Integrity:	permet au destinataire, ou à un MTA à travers lequel passe le message, de vérifier l'absence de modification du contenu du message.
Message Sequence Integrity:	permet à l'expéditeur de fournir au destinataire la preuve que la séquence des messages a été préservée.
Services de non-réfutation:	
Non-repudiation of Origin:	fournit au(x) destinataire(s) la preuve de l'origine du message, de son contenu et de ses labels de sécurité, ce qui le(s) protégera contre toute tentative de la part de l'expéditeur de renier à tort l'envoi du message, son contenu ou de ses labels de sécurité.
Non-repudiation of Submission:	fournit à l'expéditeur la preuve que le message a été soumis, ce qui le protégera contre toute tentative de la part du MTS de renier à tort le fait que le message ait été soumis pour être délivré au(x) destinataire(s) légitime(s).
Non-repudiation of delivery:	fournit à l'expéditeur la preuve que le message a été délivré , ce qui le protégera contre toute tentative de la part du (des) destinataires(s) de renier à tort la réception du message ou son contenu.
Services de gestion de la sécurité:	
Change Credentials:	fournit, à chaque composant du MHS (UA,MTA,MS), la possibilité de changer ses "pièces d'authentification" ("credentials"), contenues dans un composant quelconque du MHS.
Services de sécurité concernant la catégorisation d'un message:	
Message security labelling:	fournit la possibilité de catégoriser un message, en indiquant sa sensibilité, laquelle détermine la manipulation du message dans la ligne de la politique de sécurité.

Ces services de sécurité sont eux-mêmes basés sur des éléments de sécurité. Il faut bien faire la distinction entre services de sécurité et éléments de sécurité. Le service de sécurité est une opération offerte par le système (ex: service d'authentification de l'origine du message,...). L'élément de sécurité, quant à lui, est une fonction ou une capacité particulière du système sur lequel se base les services de sécurité. Ces éléments de sécurité manipulent des structures de données définies en ASN.1. Un service de sécurité utilise un ou plusieurs éléments de sécurité.

V.2.2. Description détaillée de quelques services de sécurité.

V.2.2.1. Structures de données utilisées par les mécanismes de sécurité.

Pour l'étude des structures de données utilisées par les mécanismes de sécurité, nous adopterons les conventions suivantes:

1°) ∀ utilisateur X et ∀ bloc de données I: **X{I}** représente le bloc de données X suivi d'une signature réalisée par l'utilisateur X sur ce bloc. Cette signature est obtenue par l'utilisation successive d'un algorithme de hachage et d'un algorithme de chiffrement à clé publique à l'aide de la clé personnelle de l'utilisateur X. Cette convention permet d'exprimer facilement des structures de données **signées** (c.-à-d. une structure de données avec une signature qui lui est accolée).

2°) ∀ bloc de données I: **H{I}** représente le résultat de l'utilisation d'une fonction de hachage sur le bloc de données I. Le résultat est également un bloc de données.

3°) ∀ utilisateur X et ∀ bloc de données I: $\mathbf{S_X\{I\}}$ représente le résultat de l'utilisation d'un algorithme de chiffrement à clé publique sur I à l'aide de la clé personnelle de l'utilisateur X. Le résultat est également un bloc de données.

4°) ∀ utilisateur X et ∀ bloc de données I: $\mathbf{V_X\{I\}}$ représente le résultat de l'utilisation d'un algorithme de chiffrement à clé publique sur I à l'aide de la clé publique de l'utilisateur X. Le résultat est également un bloc de données.

5°) ∀ utilisateur X et ∀ bloc de données signées X{I}(cf. 1°)): **GS{X{I}}** (GetSignature) est un bloc de données contenant la signature de X{I}.

6°) ∀ bloc de données I: **SC{I}** représente le résultat de l'utilisation d'un algorithme de chiffrement symétrique sur I . Le résultat est également un bloc de données.

7°) ∀ bloc de données I: **SD{I}** représente le résultat de l'utilisation d'un algorithme de déchiffrement sur I. Le résultat est également un bloc de données.

8°) ∀ utilisateur X et ∀ bloc de données I: $\mathbf{AC_X\{I\}}$ représente le résultat de l'utilisation d'un algorithme de chiffrement à clé publique sur I à l'aide de la clé publique de l'utilisateur X. Le résultat est également un bloc de données.

9°) ∀ utilisateur X et ∀ bloc de données I: $\mathbf{AD_X\{I\}}$ représente le résultat de l'utilisation d'un algorithme de chiffrement à clé publique sur I à l'aide de la clé personnelle de l'utilisateur X. Le résultat est également un bloc de données.

Au sein de l'enveloppe du message, plusieurs structures de données permettant la réalisation de services de sécurité sont disponibles, dont notamment:

-les **certificats** assurant un transport certifié des clés publiques,
-les **message-tokens** assurant un transport sécurisé des messages dans le MHS,
-les **bind-tokens** permettant une création sécurisée d'une association entre deux composantes adjacents.

Nous ne décrirons que les données les plus importantes, c.-à-d. les certificats et les message-tokens.

V.2.2.1.1. Description des certificats.

Lorsque ce sont les cryptosystèmes asymétriques qu'on utilise pour permettre la confidentialité des données et pour fournir des signatures digitales, il est indispensable de s'assurer de l'origine de la clé publique d'un tiers utilisateur avec lequel on désire communiquer. Le **certificat** est une structure de données **signée** permettant de répondre à ce besoin. Sa forme précise est définie dans X.509.

Un certificat pour une clé publique d'un utilisateur est généré par un composant externe appelé autorité certifiante, qui doit être digne de foi pour l'utilisateur. Le certificat possède la forme suivante:

CA<<A>> = CA {SgnAlg,CA,A,Ap,T^A} où

CA désigne l'autorité certifiante,
SgnAlg est l'identifiant de l'algorithme utilisé par le CA pour générer la signature du certificat,
Ap est la clé publique de A,
T^A est la période de validation du certificat.

On vérifie le certificat en hachant son contenu, et en comparant le résultat avec celui du déchiffrement de la signature à l'aide de la clé publique de l'autorité certifiante. Si les deux résultats sont égaux, alors l'utilisateur sera assuré que le certificat a bien été généré par l'autorité certifiante.

V.2.2.1.2. Description des message-tokens.

Un **message-token** est également une structure de données **signée** permettant de transférer des messages de manière sécurisée, depuis leur expéditeur vers leur(s) destinataire(s). Comme le certificat, le token est constitué d'une structure de données avec une signature digitale qui lui est accolée. La forme précise du message-token est la suivante:

A{SgnAlg,t^A,B,SgnData,EncAlg,Bp[EncData]} où

A désigne le nom de l'expéditeur du token,
B désigne le nom du destinataire,
t^A désigne la période de validation du token,
Bp[EncData] représente les données EncData **chiffrées** à l'aide de la clé publique de B,
EncAlg est l'identifiant de l'algorithme utilisé pour réaliser ce **chiffrement,**
SgnData et **EncData** sont des collections de paramètres de sécurité qui dépendent des services de sécurité fournis. Par exemple, SgnData peut comporter un champ contenant la signature hachée du contenu d'un message, permettant au destinataire de vérifier l'intégrité du contenu de celui-ci. EncData peut comporter une clé secrète utilisée pour chiffrer le contenu du message.

La structure plus précise du message-token, défini en ASN.1 est reprise à la figure 5.4.

```
message-token ::= SEQUENCE {
        type [0] INTEGER
        criticality -- missing
        value [2] MessageToken
}

MessageToken := Token := SEQUENCE {
        token-type-identifier [0] OBJECT IDENTIFIER, --"asymmetric"
        token [1] AsymmetricToken
}

AsymmetricToken := SIGNED SEQUENCE {
        signature-algorihm-id AlgorihmIdentifier,
        recipient-name ORAddressAndOrDirectoryName,
        time UTCTime,
        signed-data [0] TokenData OPTIONAL,
        encryption-algorithm-id [1] AlgorithmIdentifier OPTIONAL,
        encrypted-data [2] ENCRYPTED TokenData OPTIONAL
}

TokenData := SEQUENCE
        type [0] INTEGER -- "2" for "signed"
                         "3" for "encryped"
        value [1] MessageTokenSigned(or Encrypted)Data
}

MessageTokenSignedData := SEQUENCE {
        Content-confidentiality-Algorithm-identifier [0] AlgorithmIdendifier OPTIONAL,
        Content-integrity-check [1] ContentIntegrityCheck OPTIONAL,
        Message-security-label [2] SecurityLabel OPTIONAL,
        Proof-of-delivery-request [3] ProofOfDeliveryRequest OPTIONAL,
        Message-sequence-number [4] INTEGER OPTIONAL
}
```

```
MessageTokenEncryptedData := SEQUENCE {
        Content-confidentiality-key [0] BITSTRING OPTIONAL,
        Content-integrity-check [1] ContentIntegrityCheck OPTIONAL,
        Message-security-label [2] SecurityLabel OPTIONAL,
        Content-integrity-key [3] BITSTRING OPTONAL,
        Message-sequence-number [4] INTEGER OPTIONAL,
}
```

Figure 5.4.: Définition ASN.1 du message-token.

V.2.2.2. Le service de confidentialité du contenu du message (Message Content Confidentiality).

Ce service de sécurité peut être obtenu de deux manières différentes suivant que les paramètres utilisés sont inclus dans le message-token, ou directement dans l'enveloppe du message:

a) utilisation du **message-token**.

Le service de confidentialité du contenu est obtenu en utilisant l'élément de sécurité "**Content Confidentiality Security Element**". Cet élément de sécurité porte ce nom car il manipule la zone **encrypted-data** du message-token (voir fig. 5.4.), cette zone étant chiffrée à partir d'un algorithme à clé publique et à l'aide de la clé publique du destinataire. S'il y a plusieurs destinataires pour le message, il y aura autant de message-tokens qu'il n'y a de destinataires, chacun des tokens possédant une zone **encrypted-data** pour un destinataire bien précis. La structure de données utilisée pour obtenir la zone **encrypted-data** est le **MessageTokenEncryptedData** (voir fig. 5.4.). Dans le cas de la confidentialité du contenu du message, un champ nous intéresse plus particulièrement: le content-confidentiality-key. Il permet le transport confidentiel de la clé utilisée pour le chiffrement du contenu du message. Pour des raisons de performance (voir chapitre IV), il est vivement conseillé d'utiliser un algorithme de chiffrement symétrique (DES,...) plutôt qu'un algorithme à clé publique pour obtenir la version chiffrée du contenu du message. De plus, l'utilisation de ce champ permet de n'utiliser qu'une clé pour chiffrer le message et de n'avoir qu'une version du contenu chiffré quel que soit le nombre de destinataires. Ce service de sécurité est obtenu de la manière suivante:

$\forall$ utilisateur X et $\forall$ bloc de données I: $AD_X\{I\}$
encrypted-data=AC{MessageTokenEncryptedData}

Le déchiffrement du contenu du message se fait donc en trois étapes: la vérification de l'intégrité du message-token, le déchiffrement de la zone decrypted-data et de le déchiffrement proprement dit du contenu du message avec les données contenues dans MessageTokenEncryptedData.

La figure 5.5. résume ce que nous venons de dire en décrivant pour chaque UA les opérations qu'il peut exécuter afin de chiffrer ou déchiffrer le message.

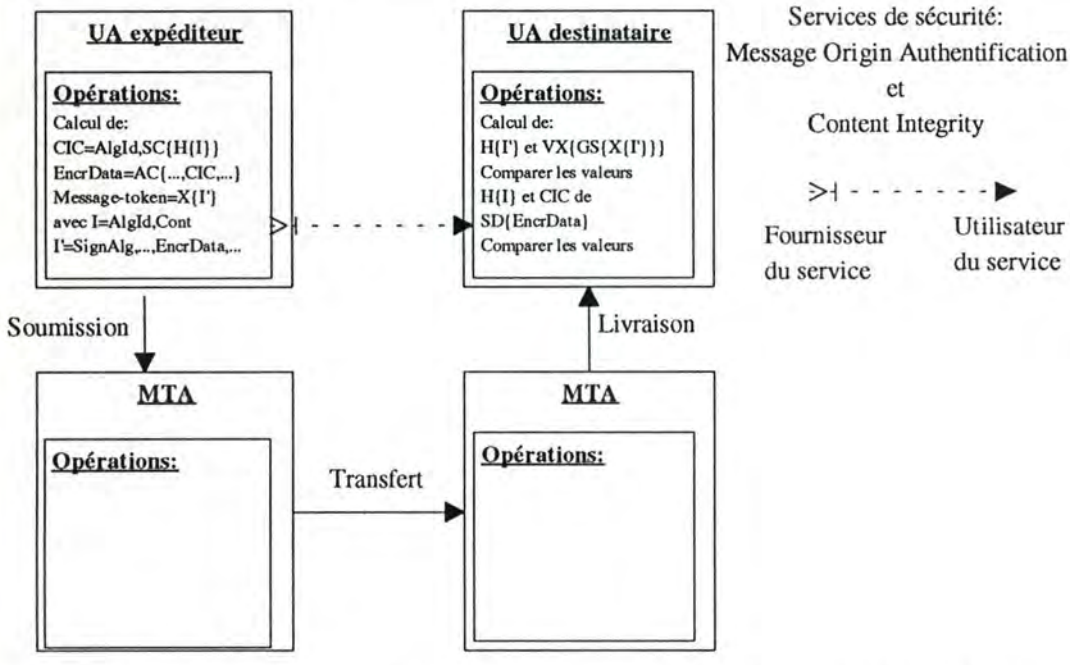

Figure 5.5.: Le service de sécurité "Content Confidentiality" utilisant l'élément de sécurité "Message Argument Confidentiality Security Element".

V.2.2.3. Les services d'authentification de l'origine du message (Message Origin Authentification) et d'intégrité du contenu du message (Content Integrity).

Trois éléments de sécurité peuvent être utilisés indépendamment pour fournir un ou deux de ces services:

a) le **Message Origin Authentification Security Element**.

Cet élément de sécurité permet à n'importe quel composant recevant ou transférant des messages d'**authentifier leur origine**, c.-à-d. le MTS-user qui en est l'expéditeur. Cet élément de sécurité correspond en fait à l'**utilisation** du champ appelé **message-origin-authentification-check (MOAC)**. C'est le MOAC qui permet de transporter l'information nécessaire permettant de vérifier l'authenticité de l'origine du message. C'est un paramètre de l'enveloppe. La forme précise du MOAC (d'après les conventions prises au paragraphe V.2.2.1.) est la suivante:

MOAC=AlgId,V_X{H{AlgId,Cont,ContId,MessSecLab}} où

X désigne le nom de l'expéditeur du MOAC,
Cont désigne le contenu du message,
ContId désigne l'identifiant du contenu du message (content-identifier),

MessSecLab désigne le "message-security-label".

Si le contenu est chiffré, alors le MOAC est calculé sur la version chiffrée du contenu du message.

La vérification de l'authenticité de l'origine du message se fait donc en deux étapes: la vérification de l'intégrité du message-token et la vérification du CIC.

L'utilisation du MOAC fournit donc un service **orienté message** d'authentification de l'origine du message, c.-à-d. qu'il permet à tous les éléments du MHS (MTA, UA,...) de vérifier l'authenticité de l'origine du message.

La figure 5.6. résume ce que nous venons de dire en décrivant pour chaque UA et chaque MTA les opérations qu'ils peuvent exécuter afin de générer ou vérifier l'authenticité de l'origine du message.

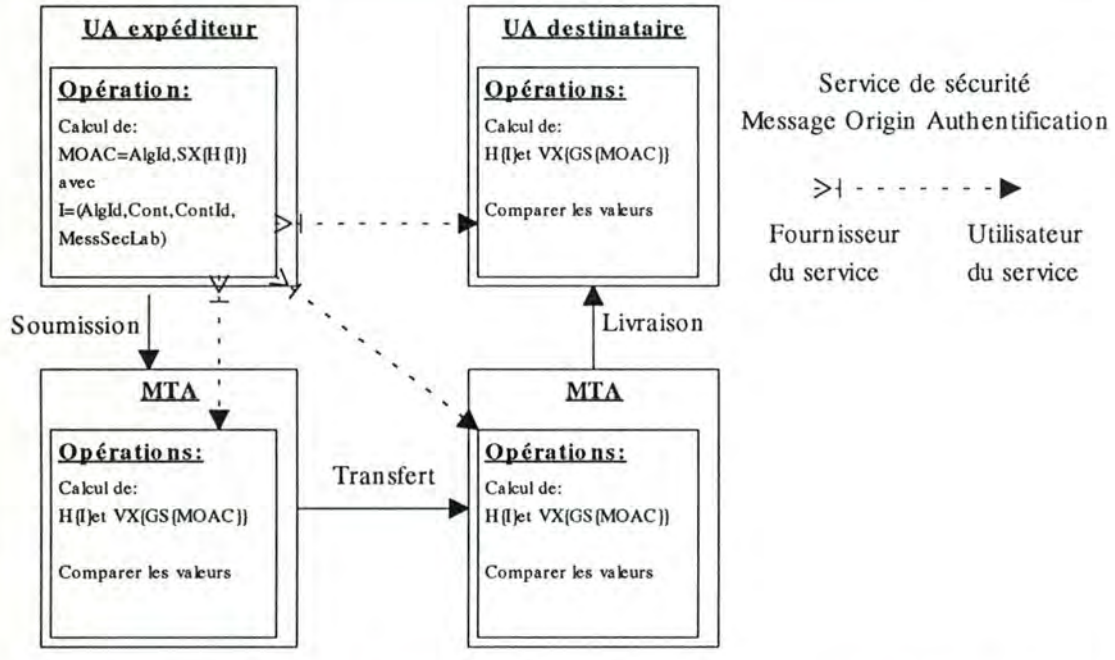

Figure 5.6.: Le service de sécurité "Message Origin Authentification" utilisant l'élément de sécurité "Message Origin Authentification Security Element", c.-à-d. le message-origin-authentification-check.

b) le **Message Argument Integrity Security Element.**

Cet élément de sécurité permet à n'importe quel composant recevant ou transférant des messages d'**authentifier leur origine**, c.-à-d. le MTS-user qui en est l'expéditeur; mais il permet également à n'importe quel composant recevant ou transférant des messages de vérifier l'**intégrité du contenu de ce message**. Cet élément de sécurité correspond en fait à l'**utilisation** du **message-token**, et en particulier à sa zone appelée **signed-data** (voir fig. 5.4.).

C'est le champ **Content-integrity-check** (**CIC**) du champ **signed-data** qui permet de transporter l'information permettant de vérifier l'authenticité de l'origine du message. Le CIC est obtenu de la manière suivante:

CIC=AlgId,V_X{H{AlgId,Cont}} où

X désigne le nom de l'expéditeur du CIC,
Cont désigne le contenu du message,
AlgId désigne l'identifiant de l'algorithme utilisé.

Le CIC est donc composé de l'identifiant de l'algorithme utilisé pour générer la signature (cette signature étant exécutée sur une suite de données composée de cet identifiant et du contenu du message) et de cette signature.

La vérification de l'authenticité de l'origine du message et/ou de l'intégrité de son contenu se fait donc en deux étapes: la vérification de l'intégrité du message-token, suivi de la vérification du MOAC.

L'utilisation de cet élément de sécurité fournit donc des services d'authentification de l'origine du message et d'intégrité du contenu du message **orientés messages**, c.-à-d. qu'il permet à tous les éléments du MHS (MTA, UA,...) de vérifier ces services. La figure 5.7. résume ce que nous venons de dire en décrivant pour chaque UA et chaque MTA les opérations qu'ils peuvent exécuter afin de générer ou vérifier l'authenticité de l'origine du message.

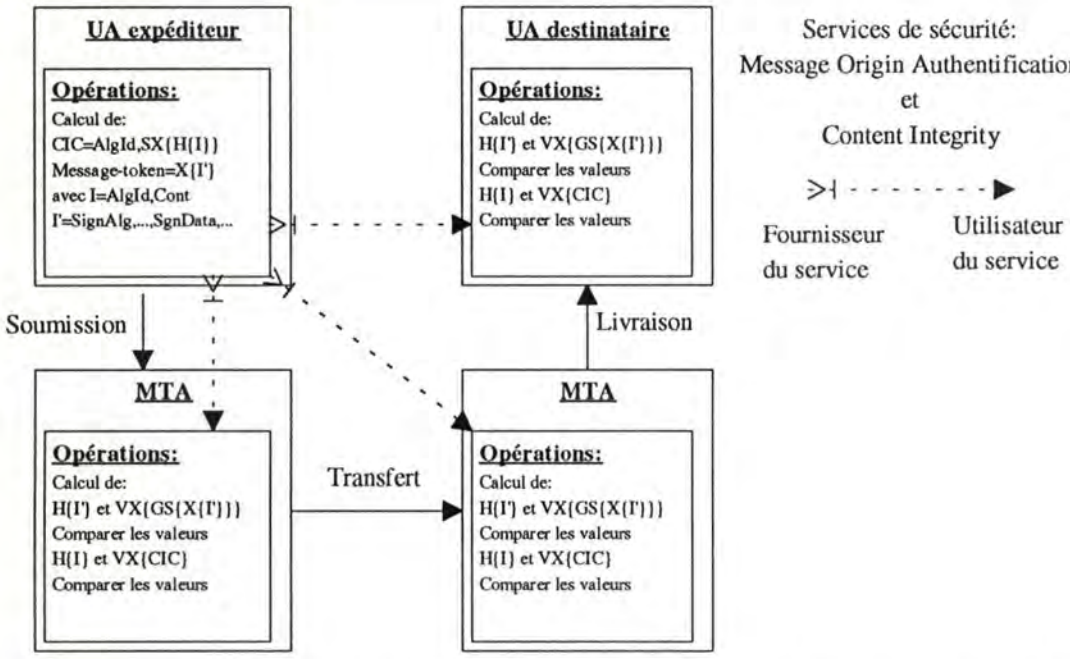

Figure 5.7.: Les services de sécurité "Message Origin Authentification" et "Content Integrity" utilisant l'élément de sécurité "Message Argument Integrity Security Element", c.-à-d. le content-integrity-check contenu dans la zone signed-data du message-token.

c) le **Message Argument Confidentiality Security Element.**

Cet élément de sécurité peut également fournir l'authentification de l'origine du message et l'intégrité du contenu du message. Cette fois-ci, c'est le champ **Content-integrity-check (CIC)** de la zone **MessageTokenEncryptedData** qui est utilisé (voir fig. 5.4.). Le CIC est soit obtenu de la même manière qu'en b), soit obtenu par l'utilisation d'un algorithme de signature symétrique et d'une clé secrète qui n'est connue que des deux interlocuteurs. De plus, les données contenues dans le MessageTokenEncryptedData sont chiffrées avec la clé publique du destinataire et incluses dans la zone **encrypted-data** du message-token:

encrypted-data=AC{MessageTokenEncryptedData}

La vérification de l'authenticité de l'origine du message et/ou de l'intégrité de son contenu se fait donc en trois étapes: la vérification de l'intégrité du message-token, le déchiffrement de la zone decrypted-data et la vérification du CIC.

L'utilisation de cet élément de sécurité fournit donc des services d'authentification de l'origine du message et d'intégrité du contenu du message **orientés destinataires**, c.-à.-d. qu'elle ne permet l'utilisation de ces services pour ce message qu'aux destinataires.

La figure 5.8. résume ce que nous venons de dire en décrivant pour chaque UA les opérations qu'il peut exécuter afin de générer ou vérifier l'authenticité de l'origine du message et l'intégrité de son contenu.

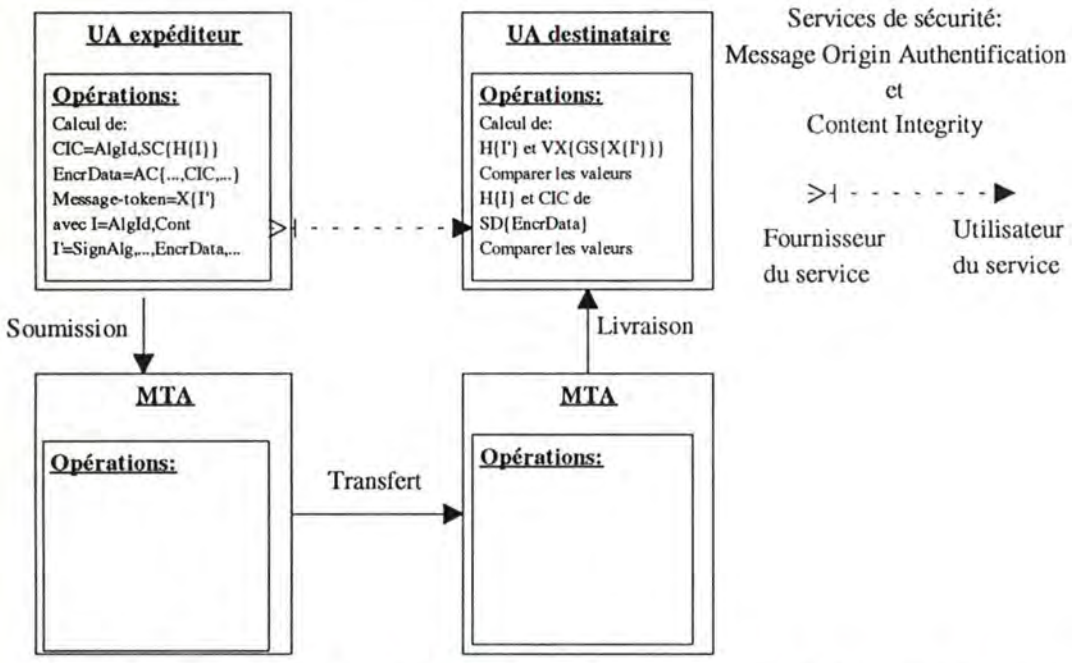

Figure 5.8.: Les services de sécurité "Message Origin Authentification" et "Content Integrity" utilisant l'élément de sécurité "Message Argument Confidentiality Security Element", c.-à-d. le content-integrity-check contenu dans la zone encrypted-data du message-token.

V.2.2.4. Les services d'authentification de l'origine du rapport de livraison/non-livraison (Report Origin Authentification).

Ce service de sécurité est fourni par l'utilisation du **Report Origin Security Element**. Cet élément de sécurité correspond en fait à l'**utilisation** du champ appelé **report-origin-authentification-check (ROAC)**. C'est le ROAC qui permet de transporter l'information nécessaire permettant de vérifier l'authenticité de l'origine du rapport. C'est un paramètre de l'enveloppe du rapport. La forme précise du ROAC est la suivante:

ROAC=AlgId,V_X{H{AlgId,ContId,MessSecLab,per-recipient}} où

X désigne le nom du MTA expéditeur du ROAC,
ContId désigne l'identifiant du contenu du message (content-identifier),
MessSecLab désigne le "message-security-label",
per-recipient contient un ensemble de données pour chaque destinataire.

Notons que des données concernant la livraison (date de livraison, type de MTS-user) ou des données concernant la non-livraison peuvent être incluses dans les données à signer.

Si le contenu est chiffré, alors le ROAC est calculé sur la version chiffrée du contenu du message.

L'utilisation du ROAC fournit donc un service **orienté message** d'authentification de l'origine du message, c.-à-d. qu'il permet à tous les éléments du MHS (MTA, UA,...) de vérifier l'authenticité de l'origine du message.

La figure 5.9. résume ce que nous venons de dire en décrivant pour chaque UA et chaque MTA les opérations qu'ils peuvent exécuter afin de générer ou vérifier l'authenticité de l'origine du message.

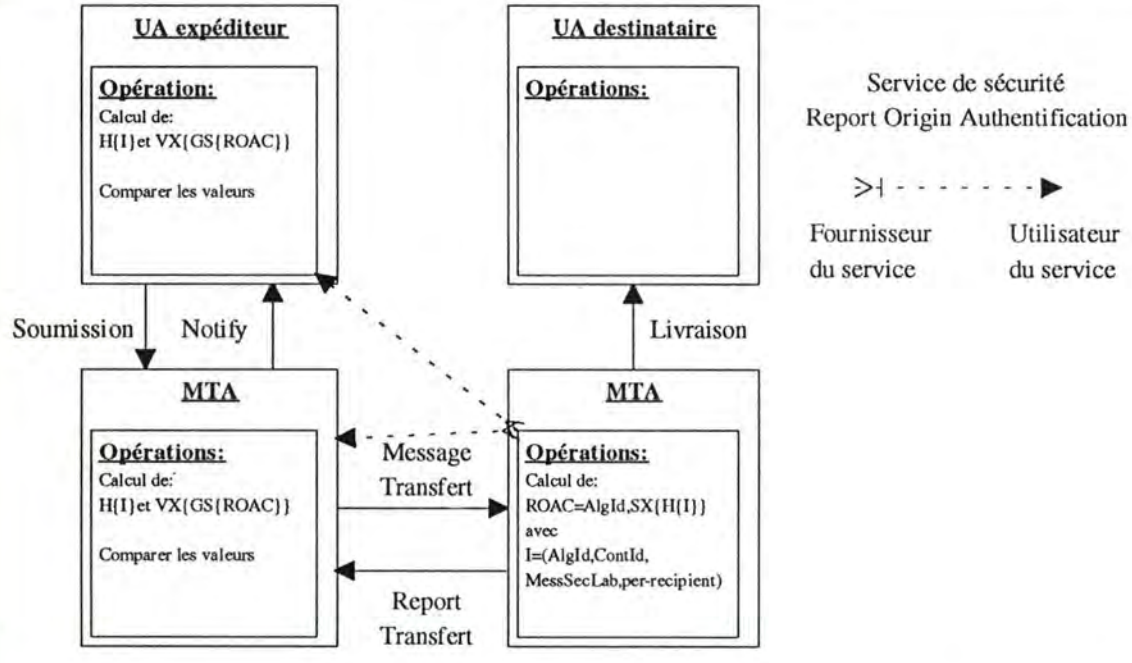

Figure 5.9.: Le service de sécurité "Message Origin Authentification" utilisant l'élément de sécurité "Message Origin Authentification Security Element", c.-à-d. le message-origin-authentification-check.

V.2.2.5. Les services fournissant la preuve de livraison du message (Proof of delivery) et de non-réfutation de cette livraison (Non-repudiation of delivery).

La preuve de livraison permet à l'expéditeur du message de s'assurer que le message a bien été délivré à ses destinataires. La non-réfutation de la livraison apporte un élément supplémentaire à savoir qu'elle empêche les destinataires de réfuter la livraison du message. Ces services se réalisent en deux temps. D'abord, l'expéditeur envoie une demande de preuve de livraison (proof-of-delivery-request) qui peut être soit un élément de l'enveloppe, soit encore un élément du message-token. Ensuite, lorsque le destinataire reçoit le message, le MTA qui a délivré ce message renvoie à l'expéditeur une signature digitale, appelée Proof-of-

delivery-check, laquelle est transportée par un rapport de livraison. La forme précise du Proof-of-delivery-check est la suivante:

Proof-of-delivery-check = AlgId,
V_X{H{AlgId,DelTime,recip,OiRecip,content,contId,MSubId,MessSecLab}} où

X désigne le nom du MTA expéditeur du Proof-of-delivery,
AlgId désigne l'identifiant de l'algorithme utilisé,
DelTime désigne la date et l'heure de la livraison du message au UA destinataire,
recip désigne l'utilisateur à qui on délivre le message,
OiRecip désigne le nom du destinataire légitime,
content désigne le contenu du message,
ContId désigne l'identifiant du contenu du message (content-identifier),
MSubId désigne le l'identifiant du message soumis,
MessSecLab désigne le "message-security-label".

Bien que le rapport soit généré par le MTS, c'est le MTS-user qui génère le Proof-of-delivery-check.

Si le contenu est chiffré, alors le Proof-of-delivery est calculé sur la version chiffrée du contenu du message.

L'utilisation du Proof-of-delivery-check fournit donc un service **orienté message** de preuve de livraison, c.-à-d. qu'il permet à tous les éléments du MHS (MTA, UA,...) de vérifier l'authenticité de l'origine du message.

La figure 5.11. résume ce que nous venons de dire en décrivant pour chaque UA et chaque MTA les opérations qu'ils peuvent exécuter afin de générer ou vérifier la preuve de livraison.

V.2.2.6. Position des services de sécurité par rapport aux couches UAL et MTL.

Les services de sécurité du MHS peuvent être classés de deux façons différentes:

- les **services de sécurité orientés message**: ce sont les services de sécurité que nous venons de voir et qui concernent la protection de bout-à-bout des messages (confidentialité, intégrité, authentification,...). Ces services sont pris en compte par la couche UA car elles font intervenir les éléments finaux de la communication (c.-à-d. les UAs). Les éléments de sécurité réalisant ces services utilisent des extensions de l'enveloppe, tels que les message-tokens, certificats,...

- **les services de sécurité orientés association**: nous n'avons pas vu d'exemples de ces services, mais nous pouvons dire qu'il s'agit de services de sécurité concernant la connexion sécurisée entre deux éléments du MHS (connexion sécurisée entre UA et MTA). Ces services sont pris en compte par la couche MT car elles font intervenir les éléments intermédiaires de la communication (UA-MTA, MTA-MTA, ...). Les éléments de sécurité réalisant ces services utilisent des paramètres dans l'opération de connexion à un élément ("bind operation" dans les protocoles d'association), tels que les bind-tokens, credentials,...

Chapitre VI:

Analyse des besoins en sécurité pour le logiciel EAN.

VI.1. Introduction.

Une description en langage ALBERT est l'exemple type d'un modèle de sécurité. En effet, non seulement l'analyste qui spécifie l'agent a le pouvoir de décider si une information sera ou ne sera pas exportée, mais en plus, un canal logique constitué de deux filtres successifs permet une exportation des informations entre les deux agents hautement sécurisée. Ces filtres correspondent en fait aux contraintes d'exportation et de perception. Par ailleurs, aucune fuite d'information n'est à craindre dans le cas où l'information n'est pas exportée.

Ce système est donc en théorie tellement sûr qu'il ne devrait pas être nécessaire de spécifier des services de sécurité. En pratique, néanmoins, on est amené à constater que l'utilisation d'un système spécifié en langage ALBERT n'est pas autant à l'abri des risques d'imposture qu'espéré. C'est ainsi qu'il nous semble malgré tout intéressant d'envisager les attaques potentielles au système et de proposer une spécification en ALBERT d'un serveur de sécurité.

VI.2. Détermination d'attaques potentielles à la spécification ALBERT de EAN.

Parmi les attaques décrites dans X.400-88, nous envisagerons successivement pour le système spécifié au chapitre III la mascarade, la fuite d'information et la modification de messages.

-La **mascarade** peut être envisagée si l'analyste déclare une action permettant de modifier l'instance UA-accessible (II.1.2.). Cette action, appelée Change-UA-accessible, permettra à un utilisateur illégitime de se connecter à l'UA sous l'identité d'un utilisateur légitime.

-La **fuite d'information** peut être envisagée si l'analyste, pour une raison quelconque, décide d'exporter les populations Stored-messages (II.1.3.) ou MTA-messages (II.1.4.) vers les agents User et que par ailleurs, les contraintes sur l'information d'état et sur la perception d'état sont telles qu'elles permettent à l'utilisateur de percevoir ces populations à un moment donné. La fuite d'information peut également être envisagée si l'analyste, pour une raison quelconque, décide d'exporter les actions SUBMIT-Req, Message-transfert ou DELIVER-Ind,... vers les agents User et que par ailleurs, les contraintes d'information sur les actions et de perception sur les actions sont telles qu'elles permettent à l'utilisateur de percevoir ces actions à un moment donné.

-Tel que nous avons spécifié les protocoles X.400 au chapitre III, la **modification des messages** y est difficilement envisageable. En effet, ces protocoles se basent sur la perception d'actions qui transfèrent les messages sous forme de paramètres. Or ces paramètres sont, par définition, non modifiables. Bien que l'exemple d'attaque que nous allons décrire ci-après (fig. 6.1.) semble être à première vue improbable lorsqu'elle est

spécifiée en langage ALBERT, envisageons la possibilité d'une telle attaque. Supposons qu'un imposteur soit capable de percevoir l'action SUBMIT-Req exécutée par l'UA et, par la même occasion, qu'il empêche la perception de celle-ci par le MTA. En possession du message, l'imposteur pourra le modifier à sa guise. Faisons une deuxième supposition: l'imposteur envoie au MTA le message modifié sous l'identité de l'UA. Cette attaque se révèle être en fait une double mascarade et il nous semble que ce soit la seule façon de décrire une modification d'un paramètre d'action de type message. La modification de messages contenus dans la population Stored-messages de l'UA ou dans la population MTA-messages du MTA peut également être envisagée si des actions permettant de le faire ont été, pour une raison quelconque, déclarées par l'analyste.

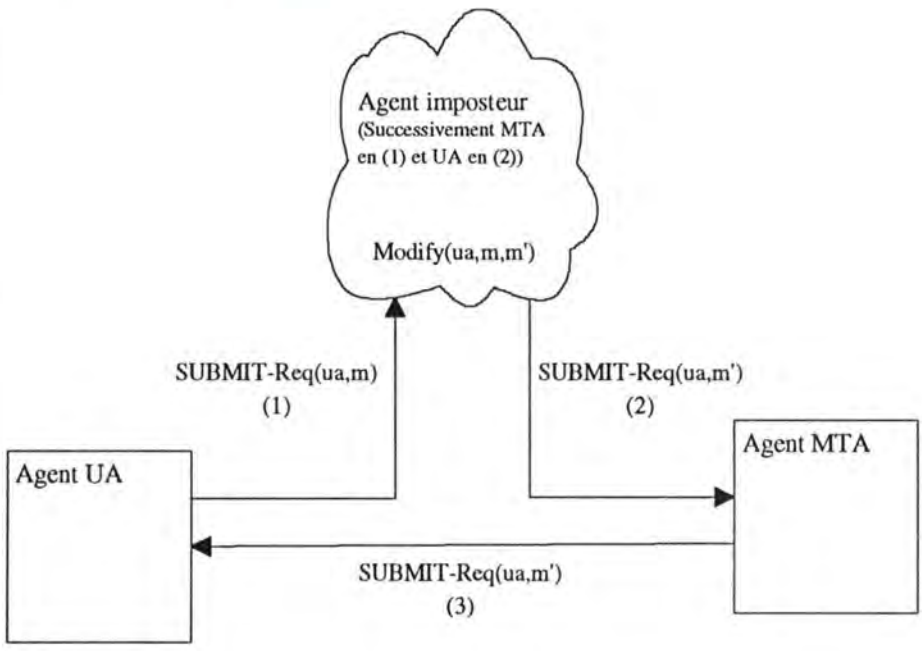

Figure 6.1.: Modélisation possible en ALBERT d'une modification de message par double mascarade.

VI.3. Buts du système.

L'établissement des services de sécurité permet d'augmenter la sécurité de la messagerie électronique entre utilisateurs. Ceci entraîne donc que les buts du système devront être adaptés aux nouveaux besoins des utilisateurs qui en résultent:

* Un message sécurisé stocké dans "Stored-messages" d'un UA et reçu par cet UA, a du être stocké dans le MTA local à cet UA. Il n'y a eu aucune modification du message-token (intégrité du message token, cf. chapitre V.).

us: USER,ua: UA, mta: MTA, mid: MESS-ID, m: MESSAGE,
us.UA-accessible = ua $\wedge$ ua.Stored-messages[mid] = m $\wedge$ In-out(ua.UA-Header[mid]) = 'in' $\wedge$
$\exists$ mt:mt $\in$ Message-token(Enveloppe(m)) $\wedge$ Card(Message-token(m)) = 1 $\wedge$
$\Rightarrow$ mta: mta.MTA-messages[m] = ua $\wedge$ mt $\in$ Message-token(Enveloppe(m))

* Un message stocké dans un MTA et devant être transféré ou délivré, a du être stocké soit dans un autre MTA, soit dans le "Stored-messages" de l'UA expéditeur. Il n'y a eu aucune modification du message-token (intégrité du message token).

mta: MTA, dest: DEST, m: MESSAGE,
mta.MTA-messages[m] = dest ∧ ∃ mt: mt ∈ Message-token(Enveloppe(m))⇒
(♦ ∃ mta':MTA, mt: MESS-TOKEN: mta'.MTA-messages[m] = mta ∧
mt ∈ Message-token(Enveloppe(m))) ∨
(♦ ∃ ua: UA,mid: MESS-ID,us: USER: ua.Stored-messages[mid] = m ∧
In-out(ua.UA-Header[mid] = 'out' ∧ us.UA-accessible = ua
mt ∈ Message-token(Enveloppe(m))) ∨
)

Ce seront les messages-tokens (cf. chapitre V) qui transporterons toutes les informations utiles à la sécurité du système de messagerie électronique. l'introduction des messages-tokens implique la "création" d'un nouvel agent appelé sec-serv (serveur de sécurité).

Etudions dans ce qui suit la mise en place, dans le système, d'un serveur de sécurité qui permettra de répondre à ces nouveaux besoins. Nous n'approfondirons parmi les services de sécurité vus au chapitre V que les suivants:

-service assurant l'**intégrité du contenu du message** (**Content Integrity**),
-service assurant la **confidentialité du contenu du message** (**Content Confidentiality**),
-service permettant de fournir la **preuve de la soumission du message** (**Proof of Submission**),
-service permettant de fournir la **preuve de la livraison du message** (**Proof of Delivery**).

Nous proposerons, en plus de ceux-ci, un service non décrit dans X.400, et permettant d'identifier l'utilisateur se connectant à l'UA.

VI.4. Spécification en langage ALBERT d'un serveur de sécurité.

VI.4.1. Déclaration de la hiérarchie des agents.

A la hiérarchie des agents décrite au chapitre III, nous ajouterons un ensemble d'agents appelé "Sec-serv" (fig.6.2.). L'agent MHS sera dès lors composé d'un quatrième sous-agent, le Sec-serv, venant s'ajouter aux trois précédents, à savoir le User-service, le MTS et le DS.

Dans ce qui suivra, nous ne décrirons pas dans le détail tous les agents car cela serait trop fastidieux dans le cadre de ce travail. Cependant nous passerons en revue 2 cas:

-la soumission d'un message par un agent UA désirant lui adjoindre des services de sécurité,
-la vérification du respect de la sécurité d'un message contenu dans l'UA.

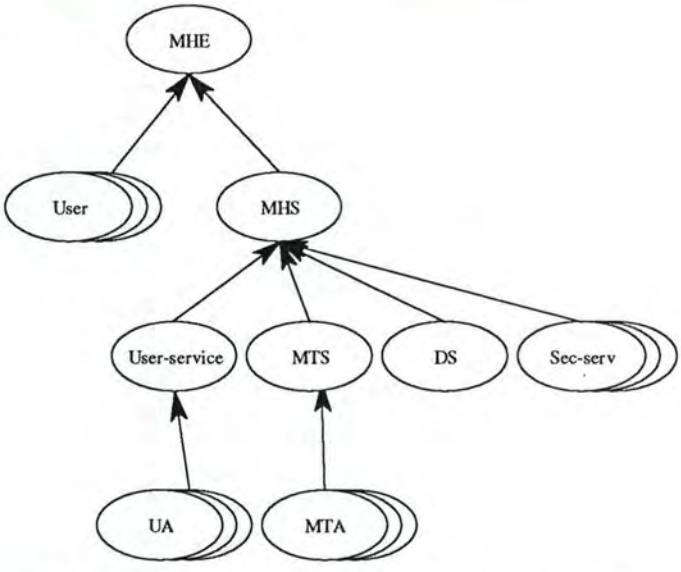

Fig 6.2.: Hiérarchie plus précise des agents du logiciel EAN.

VI.4.2 Description de la soumission d'un message sécurisé.

Par soucis de simplification, les figures 6.3. et 6.4 ne donnent la déclaration graphique en langage ALBERT que des composantes d'état et des actions supplémentaires relatives aux services de sécurité pour les agents User et UA. Il est évident que chaque agent hérite de toutes les autres actions et composantes d'état décrites précédemment au chapitre III. Pour une brève description des composantes d'état et des actions on pourra se référer à l'annexe 4. De nouveaux types abstraits de données sont définis et repris à l'annexe 5.

Décrivons maintenant, à partir de la figure 6.5. , la succession des actions réalisées ou perçues par l'agent UA lorsque celui-ci désire soumettre un message sécurisé à l'agent MTA. Pour soumettre un message sécurisé, l'agent UA perçoit ou exécute chronologiquement les actions suivantes:

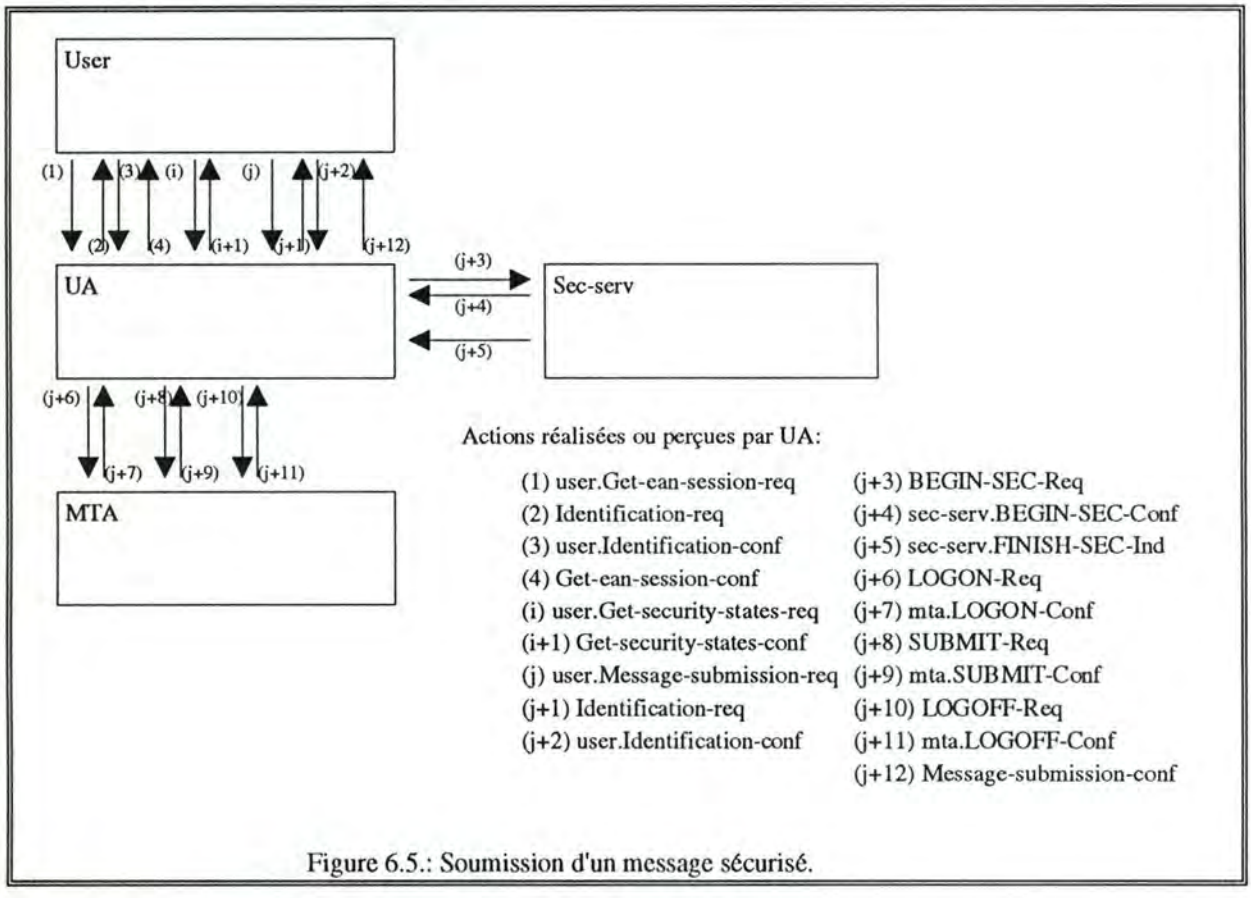

Figure 6.5.: Soumission d'un message sécurisé.

-(1) **Get-ean-session-req**: cette action est exécutée par l'utilisateur désirant obtenir une session avec l'agent UA,

-(2) **Identification-req**: exécutée par l'agent UA, cette action à pour but de demander à l'agent User de s'identifier. Ceci afin d'éviter les impostures,

-(3) **Indentification-conf**: exécutée par l'utilisateur, cette action permet à celui-ci de décliner son identité, soit par simple mot de passe, soit par un autre protocole d'identification,

-(4) **Get-ean-session-conf**: exécutée par l'UA, cette action permet, après qu'il y ait eu vérification de l'identité de l'utilisateur, de confirmer la demande de connexion,

Remarque: à ce niveau, une série d'autres actions peuvent être exécutées avant que l'utilisateur désire d'envoyer le message qui se trouve dans l'instance Draft-message.

-(i) **Get-security-state-req**: exécutée par l'utilisateur, cette action permet à celui-ci de spécifier quels seront les services de sécurité qu'il désire adjoindre au message lorsqu'il sera soumis. Cela est réalisé grâce au paramètre de type SEC-STATES. Ce type abstrait de données est un produit cartésien dont le champ de type SECREQSTATE est utilisé dans ce cas. SECREQSTATE est également un produit cartésien dont la structure est donnée à l'annexe 5 et qui permet, grâce aux valeurs "REQUESTED" et "NOTREQUESTED", de sélectionner les services adéquats,

-(i+1) **Get-security-states-conf**: exécutée par l'UA, cette action confirme le souhait qu'a l'utilisateur d'adjoindre les services de sécurité demandés, lorsque le message contenu dans l'instance Draft-message sera soumis au MTA. L'instance Draft-sec-states prendra la valeur contenue dans le paramètre de l'action Get-security-states-conf,

-(j) **Message-submission-req**: exécutée par l'utilisateur, cette action à pour but de demander à l'UA de soumettre au MTA le message contenu dans l'instance Draft-message,

-(j+1) **Identification-req**: cf.(2),

-(j+2) **Identification-conf**: cf. (3). (j+1) et (j+2) ont pour but de se réassurer à ce stade de l'identité de l'expéditeur du message,

-(j+3) **BEGIN-SECREQ-Req**: exécutée par l'UA, cette action a pour but de demander à l'agent serveur de sécurité de réaliser les opérations permettant d'obtenir un message sécurisé suivant les demandes de l'utilisateur,

-(j+4) **BEGIN-SECREQ-Conf**: exécutée par le serveur de sécurité, cette action a pour but de signaler à l'UA que sa demande a été acceptée,

-(j+5) **FINISH-SECREQ-Ind**: exécutée par le serveur de sécurité, cette action a pour but de renvoyer à l'UA le message sécurisé selon les souhaits de l'utilisateur. Ce message prendra la place de celui qui était contenu dans l'instance Draft-message. Une autre instance, appelée Draft-sec-results, prendra comme valeur celle du paramètre de même type de l'action FINISH-SECREQ-Ind. Cette instance contient les différents types d'erreurs qui auraient pu survenir aux différentes phases relatives au calcul cryptographique permettant de fournir les services de sécurité demandés,

-les étapes (j+6) **LOGON-Req**, (j+7) **LOGON-Conf**, (j+8) **SUBMIT-Req**, (j+9) **SUBMIT-Conf**, (j+10) **LOGOFF-Req** et (j+11) **LOGOFF-Conf** ont déjà été décrites au chapitre III. A ce stade, le message sécurisé a été soumis au MTA,

-(j+12) **Message-submission-conf**: exécutée par l'agent UA, cette action permet de confirmer à l'utilisateur l'acceptation de la soumission du message sécurisé. Celui-ci passe du Draft-message vers le tableau Stored-messages à une position dont la valeur est contenue dans l'instance Draft-pos.

Remarque:

Notons que, par soucis de simplification, nous n'avons décrit ni le dialogue entre le serveur de sécurité et l'agent UA, ni celui entre l'agent UA et l'agent User, ni celui entre l'agent UA et l'agent DS. Mais signalons que ces dialogues se déroulent entre les phases (j+4) et (j+5) et qu'ils ont pour but de fournir au serveur de sécurité les différents types d'algorithmes à utiliser ainsi que leurs clés, aux différentes phases cryptographiques nécessaires aux services de sécurité. Nous décrirons ces dialogues plus loin, dans les spécifications ALBERT du serveur de sécurité.

VI.4.3 Description générale de la vérification du respect de la sécurité d'un message contenu dans l'UA.

Décrivons, à partir de la figure 6.6., la succession chronologique des actions réalisées ou perçues par l'agent UA lorsque l'utilisateur désire vérifier le respect de la sécurité d'un message contenu dans la population Stored-messages de son UA:

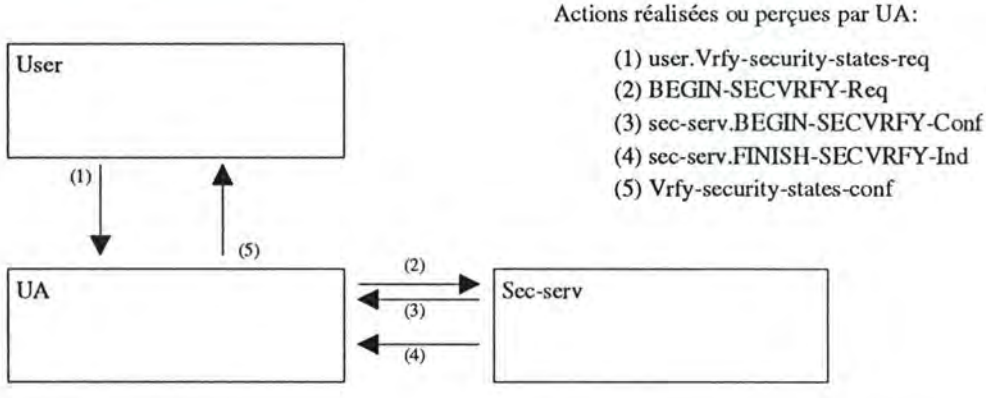

Figure 6.6.: Vérification de la sécurité d'un message

-(1) **Vrfy-security-states-req**: exécutée par l'utilisateur, cette action a pour but de demander à l'UA de vérifier le respect de la sécurité du message contenu dans l'instance Draft-message,

-(2) **BEGIN-SECVRFY-Req**: exécutée par l'UA, cette action a pour but de demander à l'agent serveur de sécurité de réaliser les opérations permettant de vérifier le respect de sécurité du message transmis comme paramètre,

-(3) **BEGIN-SECVRFY-Conf**: exécutée par le serveur de sécurité, cette action a pour but de signaler à l'UA que sa demande a été acceptée,

-(4) **FINISH-SECVRFY-Ind**: exécutée par le serveur de sécurité, cette action a pour but de renvoyer à l'agent UA les résultats de la vérification du respect de la sécurité du message demandée. Ces résultats sont communiqués via le paramètre de type SEC-STATE, et plus précisément par le champ de type SECVRFYSTATES. La valeur de ce champ sera placée dans la population Security-states à une position dont la valeur est contenue dans l'instance Draft-pos. Remarquons que l'acte de vérification de sécurité fournit des informations de types différents selon que les services sont demandés par l'expéditeur ou par le destinataire du message. Par exemple, dans notre modèle de spécification, l'UA expéditeur peut s'assurer de tous les services de sécurité demandés. Ce n'est par contre pas le cas pour le destinataire. Par exemple, l'UA destinataire se fait délivrer un message par un expéditeur (en excluant la possibilité que le destinataire s'envoye à lui-même des messages et qu'il serait donc ainsi son propre expéditeur), il lui est impossible de demander au service de sécurité de lui fournir la preuve de livraison. En effet, celui-ci ne contient aucune information concernant le rapport de livraison. Cependant, le serveur de sécurité peut fournir, et à l'expéditeur, et au destinataire tous les types de services de sécurité que l'expéditeur avait demandés. C'est en tenant compte de cela que nous avons défini le type abstrait de données SECVRFYSTATES. Celui-ci est une union de deux types OR-VRFY-ST et RECIP-VRFY-ST. Pour une

définition plus complète de ces types, nous renvoyons le lecteur à l'annexe 5. Notons également que les erreurs survenues dans le serveur de sécurité aux différentes phases relatives au calcul cryptographique, permettant de vérifier le respect de la sécurité du message, seront stockées dans la population Sec-results, à une position dont la valeur se trouve contenue dans l'instance Draft-pos,

-(5) **VRFY-security-states-conf**: exécutée par l'agent UA, cette action permet de confirmer à l'utilisateur la vérification du respect de la sécurité du message contenu dans l'instance Draft-message.

VI.4.4. Specification en ALBERT du service de sécurité.

VI.4.4.1. Description générale du serveur de sécurité.

La figure 6.7. décrit la partie générale du serveur de sécurité. Pour expliquer cette description, reprenons les deux exemples vus au VI.4.2 et VI.4.3. A la figure 6.9. est reprise en détail la spécification des contraintes pour l'agent Sec-serv, le serveur de sécurité. Dans celle-ci, le lecteur trouvera au début de certains commentaires des symboles dont les significations sont les suivantes: EA: Effects of Actions, C: Causality, AP: Action Perception, AI: Action Information.

VI.4.4.1.a. Soumission d'un message sécurisé au MTA (cf. fig 6.5.).

Lorsque l'UA décide de soumettre le message au MTA et que l'utilisateur avait préalablement formulé le souhait de sécuriser ce message, il exécute l'action BEGIN-SECREQ-Req. Cette action sera perçue par l'agent Sec-serv, lequel contient dans sa population Authorized-users l'identité de cet UA. Cette action BEGIN-SECREQ-Req a pour effet d'initialiser différentes populations du serveur (fig. 6.9., EA1):

-elle choisit un numéro d'identification qui déterminera, une fois pour toute, la position au sein des différentes populations des différentes valeurs concernées par les effets des actions,

-elle place le message reçu en paramètre dans la population SS-messages,

-elle place la valeur de type SEC-STATES reçue en paramètre dans la population Security-states,

-elle initialise le champ Current-ss de la population Sec-process contenant habituellement l'ensemble des services de sécurité en cours d'exécution ({} = ensemble vide),

-elle initialise le champ SS-done et le champ SS-to-be-done de la population Sec-process, contenant habituellemnt et respectivement l'ensemble des services de sécurité devant être réalisés et l'ensemble des services de sécurité ayant déjà été réalisés.

A partir du moment ou l'agent Sec-serv confirme la demande (BEGIN-SECREQ-Conf), plusieurs étapes peuvent être réalisées, dont certaines simultanément (fig. 6.9., C1):

-**Begin-proof-of-submission-req** suivit de **Finish-proof-of-submission-req** délimitent les actions devant se dérouler ultérieurement si l'utilisateur avait demandé que le service de sécurité Proof-of-submission soit inclu au message,

-**Begin-proof-of-delivery-req** suivit de **Finish-proof-of-delivery-req** délimitent les actions devant se dérouler ultérieurement si l'utilisateur avait demandé que le service de sécurité Proof-of-delivery soit inclu au message,

-**Begin-cont-integrity-req** suivit de **Finish-cont-integrity-req** délimitent les actions devant se dérouler ultérieurement si l'utilisateur avait demandé que le service de sécurité Content-integrity soit inclu au message,

-**Begin-cont-confidentiality-req** suivit de **Finish-cont-confidentiality-req** délimitent les actions devant se dérouler ultérieurement si l'utilisateur avait demandé que le service de sécurité Content-confidentiality soit inclu au message,

-**Begin-mess-token-sign suivit** de **Finish-mess-token-sign** permettent de signer le message-token lequel a été créé et modifié au cours des différentes phases et est contenu dans l'instance Intermediate-mess-token,

-**FINISH-SECREQ-Ind** renvoye, comme décrit au VI.4.2., le message sécurisé à l'agent UA et désalloue les éléments dans les différentes populations utilisées.

Ces services de sécurité sont exécutés selon la demande. Notons que les étapes permettant de fournir les services de sécurité Proof-of-submission et Proof-of-delivery peuvent être exécutées simultanément, alors que les étapes permettant de fournir les services de sécurité Content-integrity et Content-confidentiality doivent être exécutées successivement après les deux étapes précédentes et, bien sûr, avant l'étape de signature du message-token. Pour augmenter la sécurité du système, l'étape Content-integrity doit impérativement être exécutée avant l'étape Content-confidentiality [MWR89a]. En effet, mieux vaut signer un message non chiffré qu'un message signé, car si le message est chiffré avant d'être signé, la réfutation éventuelle ne pourra pas porter sur le message clair.

VI.4.4.1.b. Vérification du respect de la sécurité d'un message (cf. fig. 6.6.).

Après les quatre étapes successives suivantes (fig.6.9., C2):

-**BEGIN-SECVRFY-Req** (initialisation),
-**BEGIN-SECVRFY-Conf** (confirmation de l'initialisation),
-**Begin-mess-token-vrfy** suivit de **Finish-mess-token-vrfy** permettant la vérificaion de l'intégrité du message-token,

une autre série d'étapes successives vont être réalisées, dont certaines simultanément:

-**Begin-cont-confidentiality-vrfy** suivit de **Finish-cont-confidentiality-vrfy** délimitent les actions devant se dérouler ultérieurement si l'expéditeur avait demandé le service de sécurité Content-confidentiality,

-**Begin-cont-integrity-vrfy** suivit de **Finish-cont-integrity-vrfy** délimitent les actions devant se dérouler ultérieurement si l'expéditeur avait demandé le service de sécurité Content-integrity,

-**Begin-proof-of-submission-vrfy** suivit de **Finish-proof-of-submission-vrfy** délimitent les actions devant se dérouler ultérieurement si l'expéditeur avait demandé le service de sécurité Proof-of-submission,

-**Begin-proof-of-delivery-vrfy** suivit de **Finish-proof-of-delivery-vrfy** délimitent les actions devant se dérouler ultérieurement si l'expéditeur avait demandé le service de sécurité Proof-of-delivery.

Ces services de vérification de la sécurité du message sont exécutés selon les exigences de l'expéditeur. Notons que les étapes permettant de fournir les services de sécurité Proof-of-submission et Proof-of-delivery peuvent être exécutées simultanément mais uniquement après les exécutions successives des étapes permettant de fournir les services de sécurité Content-confidentiality et Content-integrity. La chronologie de la succession de ces deux étapes résulte du fait que l'expéditeur aura signé le message avant de le chiffrer.

VI.4.4.2. Description approfondie du service de sécurité "Content-integrity".

La figure 6.8. décrit le serveur de sécurité plus en détail. Pour expliquer cette description, reprenons les deux exemples vus au VI.4.2. et VI.4.3.

VI.4.4.2.a. Soumission d'un message sécurisé au MTA (cf. fig 6.5.).

Décrivons la situation à partir du moment ou l'agent Sec-sevr a perçu l'action **BEGIN-SECREQ-Req** et a confirmé celle-ci. Si l'utilisateur avait demandé à ce que le service de sécurité Content-integrity soit inclu au message, alors l'identifiant de ce service doit se trouver dans l'ensemble SS-to-be-done, lequel est un champ de la population Sec-process. Dès que l'action Begin-cont-integrity-req est exécutée par le serveur de sécurité, l'identifiant du service qui y correspond est enlevé de l'ensemble SS-to-be-done et est rajouté à l'ensemble Current-ss (lequel rappelons-le contient l'ensemble des services de sécurité exécutés sur le moment même), avec l'identifiant du processus qui a été alloué pour l'exécution de ce service (fig. 6.9. EA6). A partir de ce stade, une série d'actions successives vont être réalisées (fig 6.9. C3 et C4):

-**UA-KEY-Req**: exécutée par l'agent Sec-serv, cette action a pour but de demander à l'UA la clé secrète et l'identifiant de l'algorithme permettant de calculer le **CIC** (Content-Integrity-Check), défini au chapitre V et devant être inclu dans le message-token,

-**UA-KEY-Conf**: exécutée par l'agent UA, cette action a pour effet de renvoyer la valeur de la clé secrète et de l'identifiant de l'algorithme au serveur de sécurité qui les stockera dans la population Alg-key,

-**Message-signature-generate**: exécutée par l'agent Sec-serv, cette action a pour effet de signer le contenu du message. La valeur du champ Content-integrity-check contenue dans le champ Signed-data, lui-même contenu dans le message-token stocké dans la population Intermediate-mess-token (fig 6.9. EA21) est calculée en utilisant

successivement les opérations sur les types abstraits de données Hash et Asym-crypt (cf. annexe 5), en utilisant l'identifiant de l'algorithme de chiffrement et la clé secrète de l'utilisateur, contenus tous deux dans la population Alg-key. L'identifiant de l'algorithme de chiffrement est également intoduit dansle message-token,

-**Finish-cont-integrity-req**: exécutée par l'agent Sec-sevr, cette action a pour effet de marquer la fin de ce service de sécurité. Le processus qui servait à son exécution est désalloué. La clé et l'identifiant de l'algorithme de chiffrement sont enlevés de la population Alg-key (fig 6.9. EA10).

VI.4.4.2.b. Vérification du respect de l'intégrité du message (cf. fig. 6.6.).

Décrivons la situation à partir du moment ou l'agent Sec-serv a perçu l'action **BEGIN-SECVRFY-Req** et a confirmé celle-ci. Si l'expéditeur du message avait demandé à ce que le service de sécurité Content-integrity soit inclu au message, alors l'identifiant de ce service devra se trouver dans l'ensemble SS-to-be-done après que l'action Get-sec-states ait été exécutée. Dès que l'action Begin-cont-integrity-vrfy est exécutée par le serveur de sécurité, l'identifiant du service qui y correspond est enlevé de l'ensemble SS-to-be-done et est rajouté à l'ensemble Current-ss avec l'identifiant du processus qui a été alloué pour l'exécution de ce service (fig. 6.9. EA14). A partir de ce stade, une série d'actions successives vont être réalisées (fig. 6.9. C9 et C10):

-afin de demander à l'agent User ou à l'agent DS la clé publique de la personne qui a signé le contenu du message, ainsi que l'identifiant de l'algorithme utilisé, une des deux actions suivantes sera exécutée: soit **UA-KEY-Req**, soit **DS-KEY-Req**,

-l'agent Sec-serv recevra ces deux données soit par la perception de l'action **UA-KEY-Conf**, soit par la perception de l'action **DS-KEY-Conf**,

-**Message-signature-vrfy** exécutée par l'agent Sec-serv, cette action a pour effet de vérifier la signature du contenu du message en comparant, par une opération sur le type abstrait de données ENCRYPT-DATA, les deux valeurs suivantes: celle obtenue par l'opération Hash sur le contenu du message avec celle obtenue par l'opération Asym-decrypt sur le Content-integrity-check du Signed-data, en utilisant l'identifiant de l'algorithme de déchiffrement et la clé publique de l'utilisateur, contenus tous deux dans la population Alg-key (fig. 6.9. EA21).
L'opération Compare renvoye dans le champ Content-integrity du champ Sec-vrfy-states de la population Security-states la valeur TRUE si les deux valeurs sont égales et la valeur FALSE dans le cas contraire. L'intégrité du message sera vérifié dans le cas ou cette valeur est TRUE,

-**Finish-cont-integrity-vrfy**: exécutée par l'agent Sec-serv, cette action a pour effet de marquer la fin de ce service de sécurité. Le processus qui servait à son exécution est désalloué. La clé et l'identifiant de l'algorithme de déchiffrement sont enlevés de la population Alg-key (fig 6.9. EA18).

Sec-serv AGENT CONSTRAINTS

BASIC CONSTRAINTS

Derived Components

Initial Conditions

SS-messages[-] = UNDEF
Security-states[-] = UNDEF
Sec-results[-] = UNDEF
Requested-by[-] = UNDEF
Sec-process[-] = UNDEF
Intermediate-token[-] = UNDEF
Alg-key[-] = UNDEF

LOCAL CONSTRAINTS

States Behaviour

Sec-process[i] <> UNDEF ⇒
(Sec-id(Current-ss(Sec-process[i])) ∪ SS-done(Sec-process[i])) ∩
SS-to-be-done(Sec-process[i]) = { }

Effects Of Actions

* EA1: Quand le serveur de sécurité confirme à l'UA sa demande en services de sécurité pour un message m, il stocke le message dans SS-messages, il initialise à vide l'ensemble des processus en cours d'exécution, il initialise à vide l'ensemble des services de sécurité effectués et il détermine les services à effectuer.

BEGIN-SECREQ-Conf(-,m,st):
Security-states[i] = st
SS-messages(Sec-process[i]) = m
Current-ss(Sec-process[i]) = { }
SS-done(Sec-process[i]) = { }
SS-to-be-done(Sec-process[i]) = stb with stb = Sec-to-be-done(Security-states[i])
with i = max(SS-messages)

* EA2: Quand le serveur indique à l'UA la fin du service demandé par celui-ci, il détruit toutes les informations concernant le message.

FINISH-SECREQ-Ind(-,m,st):
Security-states[i] = UNDEF
SS-messages(Sec-process[i]) = UNDEF
Current-ss(Sec-process[i]) = UNDEF
SS-done(Sec-process[i]) = UNDEF
SS-to-be-done(Sec-process[i]) = UNDEF

* EA3: Quand le serveur de sécurité confirme à l'UA sa demande de vérification du respect de la sécurité pour un message m, il stocke le message dans SS-messages, il initialise à vide l'ensemble des processus en cours d'exécution, des services de sécurité effectués et des services à effectuer.

BEGIN-SECVRFY-Conf(-,m):
Security-states[i] = { }
SS-messages(Sec-process[i]) = m
Current-ss(Sec-process[i]) = { }
SS-done(Sec-process[i]) = { }
SS-to-be-done(Sec-process[i]) = { }

Figure 6.9.: Spécification en ALBERT des contraintes sur l'agent sec-serv, un serveur de sécurité X.400.

* EA4: Quand le serveur indique à l'UA la fin du service demandé par celui-ci, il détruit toutes les informations concernant le message..

FINISH-SECVRFY-Ind(-,m,st,res):

Security-states[i] =UNDEF
SS-messages(Sec-process[i]) = UNDEF
Current-ss(Sec-process[i]) = UNDEF
SS-done(Sec-process[i]) = UNDEF
SS-to-be-done(Sec-process[i]) = UNDEF

* EA5: Quand le serveur commence l'exécution du service de sécurité Content Confidentiality pour un message devant être sécurisé, un processus sera alloué l'identifiant du service sera supprimé de l'ensemble des services devant encore être réalisé.

Begin-cont-confidentiality-req(i,pr):

Current-ss(Sec-process[i]) = Add(Current-ss(Sec-process[i]),pr)
with process-id(pr) = Process-allocate() ∧ Sec-id(pr) = "cc"
SS-to-be-done(Sec-process[i]) = Remove(SS-done(Sec-process[i]),"cc")

* EA6: Mis à part ce qui concerne le service Content Integrity, les effets sont les mêmes que ceux décrits pour de EA5.

Begin-cont-integrity-req(i,pr):

Current-ss(Sec-process[i]) = Add(Current-ss(Sec-process[i]),pr)
with process-id(pr) = Process-allocate() ∧ Sec-id(pr) = "ci"
SS-to-be-done(Sec-process[i]) = Remove(SS-done(Sec-process[i]),"ci")

* EA7:Mis à part ce qui concerne le service Proof Of Delivery, les effets sont les mêmes que ceux décrits pour de EA5.

Begin-proof-of-delivery-req(i,pr):

Current-ss(Sec-process[i]) = Add(Current-ss(Sec-process[i]),pr)
with process-id(pr) = Process-allocate() ∧ Sec-id(pr) = "pod"
SS-to-be-done(Sec-process[i]) = Remove(SS-done(Sec-process[i]),"pod")

* EA8: Mis à part ce qui concerne le service Proof Of Submission, les effets sont les mêmes que ceux décrits pour de EA5.

Begin-proof-of-submission-req(i,pr):

Current-ss(Sec-process[i]) = Add(Current-ss(Sec-process[i]),pr)
with process-id(pr) = Process-allocate() ∧ Sec-id(pr) = "pos"
SS-to-be-done(Sec-process[i]) = Remove(SS-done(Sec-process[i]),"pos")

* EA9: Quand le serveur de sécurité a fini l'exécution du service Content Confidentiality pour un message devant être sécurisé, le processus est désalloué et l'identifiant du services est ajouté à l'ensemble des services déjà effectué.

Finish-cont-confidentiality-req(i,pr):

Current-ss(Sec-process[i]) = Remove(Current-ss(Sec-process[i]),pr)
SS-done(Sec-process[i]) = Add(SS-done(Sec-process[i]),"cc")
Alg-key[i] = UNDEF

* EA10: Mis à part ce qui concerne le service Content Integrity, les effets sont les mêmes que ceux décrits pour de EA9.

Finish-cont-integrity-req(i,pr):

Current-ss(Sec-process[i]) = Remove(Current-ss(Sec-process[i]),pr)
SS-done(Sec-process[i]) = Add(SS-done(Sec-process[i]),"ci")
Alg-key[i] = UNDEF

* EA11: Mis à part ce qui concerne le service Proof Of Delivery, les effets sont les mêmes que ceux décrits pour de EA9.

Finish-proof-of-delivery-req(i,pr):

Current-ss(Sec-process[i]) = Remove(Current-ss(Sec-process[i]),pr)
SS-done(Sec-process[i]) = Add(SS-done(Sec-process[i]),"pod")
Alg-key[i] = UNDEF

* EA12: Mis à part ce qui concerne le service Proof Of Submission, les effets sont les mêmes que ceux décrits pour de EA9.

Finish-proof-of-submission-req(i,pr):

Current-ss(Sec-process[i]) = Remove(Current-ss(Sec-process[i]),pr)
SS-done(Sec-process[i]) = Add(SS-done(Sec-process[i]),"pos")
Alg-key[i] = UNDEF

Figure 6.9.: Spécification en ALBERT des contraintes sur l'agent sec-serv, un serveur de sécurité X.400 (suite).

* EA13: Mis à part ce qui concerne la vérification, les effets sont les mêmes que ceux décrits pour de EA5 .

Begin-cont-confidentiality-vrfy(i,pr):

Current-ss(Sec-process[i]) = Add(Current-ss(Sec-process[i]),pr)
with process-id(pr) = Process-allocate() ∧ Sec-id(pr) = "cc"
SS-to-be-done(Sec-process[i]) = Remove(SS-done(Sec-process[i]),"cc")

* EA14: Mis à part ce qui concerne la vérification, les effets sont les mêmes que ceux décrits pour de EA5

Begin-cont-integrity-vrfy(i,pr):

Current-ss(Sec-process[i]) = Add(Current-ss(Sec-process[i]),pr)
with process-id(pr) = Process-allocate() ∧ Sec-id(pr) = "ci"
SS-to-be-done(Sec-process[i]) = Remove(SS-done(Sec-process[i]),"ci")

* EA15: Mis à part ce qui concerne la vérification, les effets sont les mêmes que ceux décrits pour de EA5
.

Begin-proof-of-delivery-vrfy(i,pr):

Current-ss(Sec-process[i]) = Add(Current-ss(Sec-process[i]),pr)
with process-id(pr) = Process-allocate() ∧ Sec-id(pr) = "pod"
SS-to-be-done(Sec-process[i]) = Remove(SS-done(Sec-process[i]),"pod")

* EA16: Mis à part ce qui concerne la vérification, les effets sont les mêmes que ceux décrits pour de EA5

Begin-proof-of-submission-vrfy(i,pr):

Current-ss(Sec-process[i]) = Add(Current-ss(Sec-process[i]),pr)
with process-id(pr) = Process-allocate() ∧ Sec-id(pr) = "pos"
SS-to-be-done(Sec-process[i]) = Remove(SS-done(Sec-process[i]),"pos")

* EA17: Mis à part ce qui concerne la vérification, les effets sont les mêmes que ceux décrits pour de EA9

Finish-cont-confidentiality-vrfy(i,pr):

Current-ss(Sec-process[i]) = Remove(Current-ss(Sec-process[i]),pr)
SS-done(Sec-process[i]) = Add(SS-done(Sec-process[i]),"cc")
Alg-key[i] = UNDEF

* EA18: Mis à part ce qui concerne la vérification, les effets sont les mêmes que ceux décrits pour de EA9

Finish-cont-integrity-vrfy(i,pr):

Current-ss(Sec-process[i]) = Remove(Current-ss(Sec-process[i]),pr)
SS-done(Sec-process[i]) = Add(SS-done(Sec-process[i]),"ci")
Alg-key[i] = UNDEF

* EA19: Mis à part ce qui concerne la vérification, les effets sont les mêmes que ceux décrits pour de EA9

Finish-proof-of-delivery-vrfy(i,pr):

Current-ss(Sec-process[i]) = Remove(Current-ss(Sec-process[i]),pr)
SS-done(Sec-process[i]) = Add(SS-done(Sec-process[i]),"pod")
Alg-key[i] = UNDEF

* EA20: Mis à part ce qui concerne la vérification, les effets sont les mêmes que ceux décrits pour de EA9

Finish-proof-of-submission-vrfy(i,pr):

Current-ss(Sec-process[i]) = Remove(Current-ss(Sec-process[i]),pr)
SS-done(Sec-process[i]) = Add(SS-done(Sec-process[i]),"pos")
Alg-key[i] = UNDEF

* EA21: Quand le serveur a génèré la signature du contenu du message (CIC), il place cette signature dans le message-token

Message-signature-generate(i,res) with res= ok:

Content-integrity-check(Signed-data(Intermediate-mess-token[i])) = cic
with cic = Asym-crypt(Hash(Content(Stored-messages[i])),Alg-key[i])
Alg-id(Alg-key(Signed-data(Intermediate-mess-token[i]))) = Alg-id(Alg-key[i])

* EA22

Message-signature-generate(i,res): Cont-int-result(Seq-req-result(Sec-results[i])) = res

* EA23: Quand le serveur a vérifié la signature du contenu message (CIC), le résultat de cette vérification est stocké afin d'être communiqué ultérieurement à l'UA.

Message-signature-vrfy(i,res) with res = ok: Content-integrity(Sec-vrfy-states(Security-states[i])) = val
with val = Compare(Hash(Content(Stored-messages[i])),
Asym-decrypt(Content-integrity-check(Signed-data(Intermediate-mess-token[i])),
Alg-key[i]))

* EA24

Message-signature-vrfy(i,res): Cont-int-result(Seq-vrfy-result(Sec-results[i])) = res

Figure 6.9.: Spécification en ALBERT des contraintes sur l'agent sec-serv, un serveur de sécurité X.400 (suite).

* EA25: Quand le serveur a génèré la signature du message-token, il place cette signature dans le message-token
Mess-tok-sign(i,res) with res = ok:
Mt-signature(Intermediate-mess-token[i]) = sign
with sign = Asym-crypt(Hash(Intermediate-mess-token[i]),Alg-key[i])
Signature-alg-id(Intermediate-mess-token[i]) = Alg-id(Alg-key[i])
Mtok-result(Seq-req-result(Sec-results[i])) = res

* EA26: Quand le serveur a vérifié la signature du message-token, le résultat de cette vérification est stocké afin d'être communiqué ultérieurement à l'UA.
Mess-tok-vrfy(i,res) with res = ok: Message-token(Sec-vrfy-states(Security-states[i])) = val
with val = Compare(Hash(Intermediate-mess-token[i]),
Asym-decrypt(Mt-signature(Intermediate-mess-token[i]),Alg-key[i]))

* EA27
Mess-tok-vrfy(i,res): Mtok-result(Seq-vrfy-result(Sec-results[i])) = res

* EA28: Chiffrement du contenu du message et introduction de la clé qui a servi au chiffrement dans le message-token
Message-encrypt(i,res) with res = ok:
Content(Stored-messages[i]) = cont
with cont = Sym-crypt(Content(Stored-messages[i]),Alg-key[i])
Encryption-alg-id(Intermediate-mess-token[i]) = Alg-id(Alg-key[i])

* EA29
Message-encrypt(i,res): Cont-conf-result(Seq-req-result(Sec-results[i])) = res

* EA30: Déchiffrement du contenu du message
Message-decrypt(i,res) with res = ok:
Content(Stored-messages[i]) = cont
with cont = Sym-decrypt(Content(Stored-messages[i]),Alg-key[i])

* EA31
Message-decrypt(i,res): Cont-conf-result(Seq-vrfy-result(Sec-results[i])) = res

* EA32: Chiffrement du message-token
ED-encrypt(i,recip,res) with res = ok: Add(Message-tokens(Stored-messages[i]),mtok)
with Encrypted-data(mtok) = ASym-crypt(Encrypted-data(Intermediate-mess-token[i]),Alg-key[i])

* EA33
ED-encrypt(i,recip,res): Mtok-result(Seq-req-result(Sec-results[i])) = res

* EA34: Déchiffrement du message-token
ED-decrypt(i,res) with res = ok:
Encrypted-data(Intermediate-mess-token[i]) = ed
with ed = ASym-decrypt(Encrypted-data(Intermediate-mess-token[i]),Alg-key[i])

* EA35
ED-decrypt(i,recip,res): Mtok-result(Seq-vrfy-result(Sec-results[i])) = res

* EA36: Le serveur désire obtenir le message-token pour ces calculs
Get-mess-token(i,res) with res= ok: Intermediate-mess-token[i] = mt with mt = Get-env-mt(SS-messages[i])

* EA37: Le serveur désire connaître quels étaient les services de sécurité réclamé par l'expéditeur.
Get-sec-states(i): Security-states[i] = st with st = Get-mt-st(Intermediate-mess-token[i])

Figure 6.9.: Spécification en ALBERT des contraintes sur l'agent sec-serv, un serveur de sécurité X.400 (suite).

Causality

* C1: Actions devant survenir lorsqu'il y eu une demande pour sécuriser un message

BEGIN-SECREQ-Req(-,m,i,st) $\xrightarrow{\diamond}$
BEGIN-SECREQ-Conf(ua,i);
[
{((Begin-proof-of-submission-req(i);Finsih-proof-of-submission-req(i))
⊕ DAC) ||
((Begin-proof-of-delivery-req(i);Finish-proof-of-delivery-req(i))
⊕ DAC)};
{(Begin-cont-integrity-req(i);Finish-cont-integrity-req(i))
⊕ DAC};
{(Get-mess-token(i);
Begin-cont-confidentiality-req(i,mtok);Finish-cont-confidentiality-req(i,mtok))
⊕ DAC}
];
Begin-mess-token-sign(i);Finish-mess-token-sign(i);

FINISH-SECREQ-Ind(ua,m,-)

* C2: Actions devant survenir lorsqu'il y eu une demande vérification pour un message sécurisé

BEGIN-SECVRFY-Req(-,m,i,-,-) $\xrightarrow{\diamond}$
BEGIN-SECVRFY-Conf(ua,i);Get-mess-token(i,res)
Begin-mess-token-vrfy(i);Finish-mess-token-vrfy(i);Get-sec-states(i);
[
{(Begin-cont-confidentiality-vrfy(i);Finish-cont-confidentiality-vrfy(i)) ⊕ DAC};
{(Begin-cont-integrity-vrfy(i);Finish-cont-integrity-vrfy(i)) ⊕ DAC};
{((Begin-proof-of-submission-vrfy(i);Finish-proof-of-submission-vrfy(i))
⊕ DAC) ||
((Begin-proof-of-delivery-vrfy(i);Finish-proof-of-delivery-vrfy(i))
⊕ DAC)}
];
FINISH-SECVRFY-Ind(ua,m,-,-)

* C3: Le dialogue devra s'établir pour l'obtention d'une clé secrète de chiffrement du CIC

Begin-cont-integrity-req(i) $\xrightarrow{\diamond}$
UA-KEY-Req(ua,"CIC",-)

* C4: Dialogue pour l'obtention d'une clé. Cette clé servira à la signature du contenu du message qui devra avoir lieu.

Begin-cont-integrity-req(i);
UA-KEY-Req(ua,"CIC",-);
ua.UA-KEY-Conf(ss,cic-alg-key)
$\xrightarrow{\diamond}$
Message-signature-generate(i,res);
Finish-cont-integrity-req(i)

* C5: Le dialogue devra s'établir pour l'obtention d'une clé secrète de chiffrement du CIC

Begin-cont-confidentiality-req(i) $\xrightarrow{\diamond}$
UA-KEY-Req(ua,"CC",-)

* C6: Dialogue pour l'obtention d'une clé. Cette clé servira à la signature du contenu du message qui devra avoir lieu.

Begin-cont-confidentiality-req(i);
UA-KEY-Req(ua,"CC",-);
ua.UA-KEY-Conf(ss,cc-alg-key);
$\xrightarrow{\diamond}$
Message-encrypt(i,res);
∀ recip ∈ Recip(Enveloppe(SS-messages[i])):
Begin-ed-encrypt(i,recip,res);Finish-ed-encrypt(i);
finish-cont-confidentiality-req(i)

Figure 6.9.: Spécification en ALBERT des contraintes sur l'agent sec-serv, un serveur de sécurité X.400 (suite).

```
* C7
        Begin-mess-token-generate(m,-,-,-) ◊⟶
                UA-KEY-Req(ua,"MessToken",-)
* C8
        Begin-mess-token-sign(i);
        UA-KEY-Req(ua,"MessToken",-);
        ua.UA-KEY-Conf(ss,alg-key-tok)
        ◊⟶
                ∀ mtok ∈ Message-tokens(Enveloppe(SS-messages[i])):
                Mess-token-sign(i,,res);Finish-mess-token-generate(i)
* C9
        Begin-cont-integrity-vrfy(i) ◊⟶
                UA-KEY-Req(ua,"CIC",Cic-alg-id(Signed-data(Message-token(Enveloppe(m))))) ⊕
                DS-KEY-Req(Origin(Enveloppe(m)),Cic-alg-id(Signed-data(Message-token(Enveloppe(m)))))
* C10
        Begin-cont-integrity-vrfy(i)
        [{UA-KEY-Req(ua,"CIC",Cic-alg-id(Signed-data(Message-token(Enveloppe(m)))));
        ua.UA-KEY-Conf(ss,alg-key-cic)} ⊕
        {DS-KEY-Req(Origin(Enveloppe(m)),Cic-alg-id(Signed-data(Message-token(Enveloppe(m)))));
        ds.DS-KEY-Conf(ss,ua,alg-key-cic)}]
        ◊⟶
                Message-signature-vrfy(i,res)
                Finish-cont-integrity-vrfy(i)
* C11
        Begin-cont-confidentiality-vrfy(i) ◊⟶
                UA-KEY-Req(ua,"cc",Encryption-alg-id(Message-token(Enveloppe(m))))
* C12
        Begin-cont-confidentiality-vrfy(i)
        UA-KEY-Req(ua,"EDMT",-);
        ua.UA-KEY-Conf(ss,edmt-alg-key)
        ◊⟶
                ED-decrypt(i,res)
                Message-decrypt(i,res)
                Finish-cont-confidentiality-req(i)
* C13
        Begin-mess-token-vrfy(i) ◊⟶
                UA-KEY-Req(ua,"MessToken",Signature-alg-id(Message-token(Enveloppe(m)))) ⊕
                DS-KEY-Req(Origin(Enveloppe(m)),Signature-alg-id(Message-token(Enveloppe(m))))
* C14
        Begin-mess-token-vrfy(i);
        [{UA-KEY-Req(ua,"MessToken",Signature-alg-id(Message-token(Enveloppe(m))));
        ua.UA-KEY-Conf(ss,ua,alg-key-tok)} ⊕
        {DS-KEY-Req(Origin(Enveloppe(m)),Signature-alg-id(Message-token(Enveloppe(m))));
        ds.DS-KEY-Conf(ss,alg-key-tok)}]
        ◊⟶ Mess-token-vrfy(i,alg-key-tok);Finish-mess-token-vrfy(i)
* C15
        Begin-ed-encrypt-req(i) ◊⟶
                UA-KEY-Req(ua,"EDMT",-)
* C16
        Begin-ed-encrypt(i)
        UA-KEY-Req(ua,"EDMT",-);
        ua.UA-KEY-Conf(ss,edmt-alg-key)
        ◊⟶
                ED-encrypt(i,res)
                Finish-ed-encrypt(i)
```

Figure 6.9.: Spécification en ALBERT des contraintes sur l'agent sec-serv, un serveur de sécurité X.400 (suite).

Capability

*** Cap1 → Cap4**: Pour qu'un service de sécurité puisse s'exécuter pour sécuriser un message, il faut que ce sevice ait été demandé par l'UA expéditeur connu, du serveur. Ce service ne doit pas avoir déjà été exécuté sur ce message.

* Cap1

F(Begin-proof-of-submission-req(i,pr) /
Requested-by[i] <> UNDEF ∨ ¬ In(Requested-by[i],Authorized-uas) ∨
Origin(Enveloppe(m)) <> Requested-by[i] ∨
Proof-of-submission(Sec-req-states(Security-states[i])) <> 'REQUESTED') ∨
¬ In("pos",Ss-to-be-done(Sec-process[i])) ∨ In("pos",Ss-done(Sec-process[i])))

* Cap2

F(Begin-proof-of-delivery-req(i,pr) /
Requested-by[i] <> UNDEF ∨ ¬ In(Requested-by[i],Authorized-uas) ∨
Origin(Enveloppe(m)) <> Requested-by[i] ∨
Proof-of-delivery(Sec-req-states(Security-states[i])) <> 'REQUESTED') ∨
¬ In("pod",Ss-to-be-done(Sec-process[i])) ∨ In("pod",Ss-done(Sec-process[i])))

* Cap3

F(Begin-cont-integrity-req(i,pr) /
Requested-by[i] <> UNDEF ∨ ¬ In(Requested-by[i],Authorized-uas) ∨
Origin(Enveloppe(m)) <> Requested-by[i] ∨
Content-integrity(Sec-req-states(Security-states[i])) <> 'REQUESTED') ∨
¬ In("ci",Ss-to-be-done(Sec-process[i])) ∨ In("ci",Ss-done(Sec-process[i])))

* Cap4

F(Begin-cont-confidentiality-req(i,pr) /
Requested-by[i] <> UNDEF ∨ ¬ In(Requested-by(m),Authorized-uas) ∨
Origin(Enveloppe(m)) <> Requested-by(m) ∨
Content-confidentiality(Sec-req-states(Security-states[m])) <> 'REQUESTED') ∨
¬ In("cc",Ss-to-be-done(Sec-process[m])) ∨ In("cc",Ss-done(Sec-process[i])))

*** Cap5 → Cap6**: Pour qu'un service de sécurité (Proof-x) puisse s'exécuter pour vérifier la sécurité d'un message, il faut que ce sevice ait été demandé par l'UA expéditeur du message, connu du serveur. Ce service ne doit pas avoir déjà été exécuté sur ce message.

* Cap5:

F(Begin-proof-of-submission-vrfy(i,pr) /
Requested-by[i] <> UNDEF ∨ ¬ In(Requested-by[i],Authorized-uas) ∨
Origin(Enveloppe(Stored-messages[i])) <> Requested-by[i] ∨
Proof-of-submission(Sec-vrfy-states(Security-states[i])) <> 'REQUESTED') ∨
¬ In("pos",Ss-to-be-done(Sec-process[i])) ∨ In("pos",Ss-done(Sec-process[i])) ∨
Enveloppe(Stored-messages[i]) = UNDEF)

* Cap6

F(Begin-Proof-of-delivery-vrfy(i,pr) /
Requested-by[i] <> UNDEF ∨ ¬ In(Requested-by[i],Authorized-uas) ∨
Origin(Enveloppe(Stored-messages[i])) <> Requested-by[i] ∨
Proof-of-delivery(Sec-vrfy-states(Security-states[i])) <> 'REQUESTED') ∨
¬ In("pod",Ss-to-be-done(Sec-process[i])) ∨ In("pod",Ss-done(Sec-process[i])) ∨
Enveloppe(Stored-messages[i]) = UNDEF)

*** Cap7 → Cap8**: Pour qu'un service de sécurité (Content-x) puisse s'exécuter pour vérifier la sécurité d'un message, il faut que ce sevice ait été demandé par un UA, connu du serveur. Ce service ne doit pas avoir déjà été exécuté sur ce message.

* Cap7

F(Begin-Content-integrity-vrfy(i,pr) /
Requested-by[i] <> UNDEF ∨ ¬ In(Requested-by[i],Authorized-uas) ∨
Get-recip(Recip(Enveloppe(Stored-messages[i]))) <> Requested-by[i] ∨
Content-integrity(Sec-vrfy-states(Security-states[i])) <> 'REQUESTED') ∨
¬ In("ci",Ss-to-be-done(Sec-process[i])) ∨ In("ci",Ss-done(Sec-process[i])) ∨
Enveloppe(Stored-messages[i]) = UNDEF)

Figure 6.9.: Spécification en ALBERT des contraintes sur l'agent sec-serv, un serveur de sécurité X.400 (suite).

```
* Cap8
        F(Begin-Content-confidentiality-vrfy(i,pr) /
                Requested-by[i] <> UNDEF ∨ ¬ In(Requested-by[i],Authorized-uas) ∨
                Get-recip(Recip(Enveloppe(Stored-messages[i]))) <> Requested-by[i] ∨
                Content-integrity(Sec-vrfy-states(Security-states[i])) <> 'REQUESTED') ∨
                ¬ In("cc",Ss-to-be-done(Sec-process[i])) ∨ In("cc",Ss-done(Sec-process[i])) ∨
                Enveloppe(Stored-messages[i]) = UNDEF)

* Cap9
        F(Finish-proof-of-submission-req(i,pr) / ¬ In(pr,Current-ss(Sec-process[i]) ∨ Sec-id(pr) <> "pos")
* Cap10
        F(Finish-proof-of-delivery-req(i,pr) / ¬ In(pr,Current-ss(Sec-process[i]) ∨ Sec-id(pr) <> "pod")
* Cap11
        F(Finish-cont-integrity-req(i,pr) / ¬ In(pr,Current-ss(Sec-process[i]) ∨ Sec-id(pr) <> "ci")
* Cap12
        F(Finish-cont-confidentiality-req(i,pr) / ¬ In(pr,Current-ss(Sec-process[i]) ∨ Sec-id(pr) <> "cc")
* Cap13
        F(Finish-proof-of-submission-vrfy(i,pr) / ¬ In(pr,Current-ss(Sec-process[i]) ∨ Sec-id(pr) <> "pos")
* Cap14
        F(Finish-proof-of-delivery-vrfy(i,pr) / ¬ In(pr,Current-ss(Sec-process[i]) ∨ Sec-id(pr) <> "pod")
* Cap15
        F(Finish-cont-integrity-vrfy(i,pr) / ¬ In(pr,Current-ss(Sec-process[i]) ∨ Sec-id(pr) <> "ci")
* Cap16
        F(Finish-cont-confidentiality-vrfy(i,pr) / ¬ In(pr,Current-ss(Sec-process[i]) ∨ Sec-id(pr) <> "cc")
* Cap17
        F(Message-signature-generate(i,-) /
                Is-of_ENCRCONTENT(Content(Stored-messages[i])) ∨ Alg-key[i] = UNDEF
* Cap18
        F(Message-signature-vrfy(i,-) /
                ¬ Is-of_ENCRCONTENT(Content(Stored-messages[i])) ∨ Alg-key[i] = UNDEF
* Cap19
        F(Mess-tok-sign(i,-) / Alg-key[i] = UNDEF
* Cap20
        F(Mess-tok-vrfy(i,-) / Alg-key[i] = UNDEF
* Cap21
        F(Message-encrypt(i,-) /
                Is-of_ENCRCONTENT(Content(Stored-messages[i])) ∨ Alg-key[i] = UNDEF
* Cap22
        F(Message-decrypt(i,-) /
                ¬ Is-of_ENCRCONTENT(Content(Stored-messages[i])) ∨ Alg-key[i] = UNDEF
* Cap23
        F(ED-encrypt(i,-,-) /
                Is-of_ENCRMT-DATA(Encrypt-data(Intermediate-mess-tok[i])) ∨ Alg-key[i] = UNDEF
* Cap24
        F(ED-decrypt(i,-) /
                ¬ Is-of_ENCRMT-DATA(Encrypt-data(Intermediate-mess-tok[i])) ∨ Alg-key[i] = UNDEF
* Cap25
        F(Get-mess-token(i,-) /
                Stored-messages[i] = UNDEF ∨
                Card(Message-tokens(Enveloppe(Stored-messages[i]))) <> 1)
* Cap26
        F(Get-sec-states(i,-) / Stored-messages[i] = UNDEF ∨ Intermediate-mess-token[i] = UNDEF)
* Cap27
        F(UA-KEY-Req(ua,-,-) / ¬ In(ua,Authorized-uas))
```

Figure 6.9.: Spécification en ALBERT des contraintes sur l'agent sec-serv, un serveur de sécurité X.400 (suite).

COOPORATION CONSTRAINTS

Action Perception

* AP1: Le serveur ne prend pas en compte les requêtes d'un UA qui ne connaît pas.
I(ua.BEGIN-SECREQ-Req(ss,-,-) /
¬ In(ua,Authorized-uas) ∨ ss <> self)

* AP2: Le serveur ne prend pas en compte les requêtes d'un UA qui ne connaît pas.
I(ua.BEGIN-SECVRFY-Req(ss,-,-) /
¬ In(ua,Authorized-uas) ∨ ss <> self)

* AP3
I(ua.UA-KEY-Conf(ss,-,-) /
¬ In(ua,Authorized-uas) ∨ ss <> self)

* AP4
I(ua.DS-KEY-Conf(ss,-,-) / ss <> self)

State Perception

Action Information

* AI1: Le serveur confirme toujours une demande de service d'un UA connu
XK(BEGIN-SECREQ-Conf(ua',m,i,-).ua /
In(ua,Authorized-uas) ∧ Requested-by[i] <> UNDEF ∧
Origin(Enveloppe(Stored-messages[i])) = Requested-by[i] ∧ Requested-by[m] = ua ∧ ua = ua')

* AI2: Le serveur confirme toujours une demande de service d'un UA connu
XK(BEGIN-SECVRFY-Conf(ua',i,-).ua /
In(ua,Authorized-uas) ∧ Requested-by[i] <> UNDEF ∧ Requested-by[i] = ua ∧ ua = ua')

* AI3: Le serveur indique toujours la fin d'un service.
XK(FINISH-SECREQ-Ind(ua',i,-,-).ua /
In(ua,Authorized-uas) ∧ Requested-by[m] <> UNDEF ∧ Origin(Enveloppe(m)) = Requested-by[m] ∧
Requested-by[m] = ua ∧ ua = ua')

* AI4: Le serveur indique toujours la fin d'un service.
XK(FINISH-SECVRFTY-Ind(ua',m,-,-).ua /
In(ua,Authorized-uas) ∧ Requested-by[m] <> UNDEF ∧ Requested-by[m] = ua ∧ ua = ua')

*AI5
XK(UA-KEY-Req(ua',-,-).ua / ua = ua')

*AI6
XK(DS-KEY-Req(ua',-).ua / ua = ua')

State Information

Figure 6.9.: Spécification en ALBERT des contraintes sur l'agent sec-serv, un serveur de sécurité X.400 (suite).

VI.4.4.3. Perspectives d'architecture.

Voyons à présent les perspectives d'architecture que l'on peut dégager à partir de la spécification en ALBERT du serveur de sécurité. Celles-ci sont de trois types.

VI.4.4.3.1. Fonctions de haut niveau et fonctions de bas niveau.

Il apparaît dans la description ALBERT du serveur de sécurité, deux niveaux de fonctions assurant le service de sécurité. D'une part, des fonctions que l'on nommera fonctions de haut niveau, représentées dans notre description par les actions internes du serveur de sécurité et qui fournissent les services de sécurité. D'autre part, des fonctions que l'on nommera fonctions de bas niveau, représentées dans notre description par les opérations sur les types abstraits de données et qui fournissent les fonctions cryptographiques de base aux fonctions de sécurité de haut niveau.. [WRM93] propose, dans la phase de réalisation, une architecture qui tient compte de ce fait. Un design orienté objet pourrait également être envisagé pour la réalisation de ce serveur de sécurité.

VI.4.4.3.2. Définition de primitives de service entre l'UA et le serveur de sécurité.

Les actions de haut niveau et de bas niveau sont sous la responsabilité du serveur de sécurité. Un agent UA ne peut en aucun cas exécuter lui-même ces fonctions. Cependant, il peut formuler une requête au serveur de sécurité et demander qu'il exécute tel ou tel service de sécurité. Lors de la description générale de la soumission d'un message sécurisé (VI.4.2. fig. 6.6.) et de la vérification du respect de la sécurité du message, nous avions déjà pu spécifier les actions **BEGIN-SECREQ-Req**, **BEGIN-SECREQ-Conf**, **FINISH-SECREQ-Ind**, **BEGIN-SECVRFY-Req**, **BEGIN-SECVRFY-Conf** et **FINISH-SECVRFY-Ind** qui permettaient le dialogue entre l'UA et le serveur de sécurité. Lors de la spécification du serveur de sécurité, nous avons pu également isoler d'autres actions permettant à l'UA ou au DS de fournir des clés et des identifiants d'algorithme de chiffrement au serveur de sécurité.

VI.4.4.3.3. Exécutions simultanées de certains services de sécurité.

C'est la capacité que possède ALBERT de modéliser de manière déclarative des actions qui s'exécutent simultanément (cf. contraintes de causalité) qui nous a ouvert les yeux sur, d'une part la possibilité de l'exécution simultanée de certains services de sécurité et, d'autre part la possibilité que pourrait avoir le serveur de sécurité de répondre simultanément aux demandes de plusieurs UAs (plusieurs messages seront traités en même temps).

Dans la pratique, il serait possible de réaliser ce serveur en utilisant les techniques d'architecture parallèle, d'architecture distribuée,... Cependant, le langage ALBERT étant déclaratif, il ne nous donne pas de solutions algorithmiques quant à la gestion de ces exécutions simultanées. Ainsi, ce sera plutôt à la phase de design que l'on choisira une politique pour la gestion de ces exécutions simultanées.

Conclusions.

La description du serveur de sécurité par un langage formel de spécification tel que le langage ALBERT a fait ressortir de notre étude des caractéristiques fondamentales relatives à son architecture et à ses liens avec l'UA.

La première de ces caractéristiques relatives à l'architecture concerne la confirmation de la subdivision des fonctions en fonctions de haut niveau d'une part, et de bas niveau d'autre part. Aux actions décrites dans la spécification en ALBERT du serveur de sécurité correspondent des fonctions de haut niveau c.-à-d. des fonctions décrivant les fonctionnalités de base du système (les services de sécurité) et des fonctions modifiant la structure du message (signature du contenu, signature du message-token,...). Tandis qu'aux opérations sur les types abstraits de données correspondent des fonctions de bas niveau c.-à-d. des fonctions cryptographiques de base (fonctions de (dé)chiffrement à clé secrète ou à clé publique, fonctions de hachage,...).

La seconde caractéristique relative à l'architecture et qui nous a été révélée par la spécification de liens de causalité entre actions concerne la capacité qu'a le système d'exécuter des opérations de manière simultanée (en parallèle). Ainsi, ce système à la possibilité de traiter plusieurs messages en même temps et, pour chacun de ceux-ci, plusieurs services de sécurité peuvent être exécutés simultanément.

Il nous semble qu'il serait intéressant d'envisager l'implémentation d'un tel système en se référant à ces deux constatations.

La caractéristique fondamentale relative aux liens unissant l'UA et le serveur de sécurité concerne son autonomie relative. En effet, la gestion de l'exécution et l'exécution même des services de sécurité sont entièrement sous sa responsabilité. De plus, pour l'UA, le serveur de sécurité est une "boîte noire" à laquelle il n'a pas accès: il ne peut ni accéder aux informations qu'elle contient, ni exécuter une fonction qui appartient à cette boîte. Ceci contribue à augmenter la sécurité du système. Remarquons cependant que son autonomie n'est pas totale en ce sens qu'il doit impérativement répondre aux requêtes de l'UA et lui fournir les résultats de sa requête. Notons quand même qu'une faille relative aux liens unissant l'UA au service de sécurité nous est apparue: c'est en clair que l'UA communique sa clé secrète au serveur de sécurité. En définitive, le concept d'autonomie du serveur de sécurité place celui-ci sur un pied d'égalité par rapport à l'UA et au MTA.

Aux vues des résultats obtenus par la spécification en langage ALBERT, nous pouvons affirmer que ce langage s'adapte bien au domaine des télécommunications en général, et à celui de la sécurité de la messagerie électronique en particulier. Par ailleurs, c'est grâce à ce langage que nous avons pu mettre en évidence les caractéristiques fondamentales relatives à l'autonomie du serveur de sécurité décrites ci-avant. Cette propriété d'autonomie que possède le serveur de sécurité n'est en fait que l'héritage que lui confère son statut d'agent. Ce langage ne s'est cependant pas révélé adéquat dans le cas de la description de l'attaque consistant à modifier le contenu du message lors de sa soumission. Cela s'explique par le fait que toute communication entre agents se réalise par l'exécution d'une action.

Enfin, il serait intéressant lorsqu'on raffine un agent en lui ajoutant de nouvelles actions ou de nouvelles composantes d'état, d'utiliser un mécanisme d'héritage. En effet, dans le cas du serveur de sécurité, la déclaration graphique est fort chargée et une spécification par raffinements successifs en utilisant ce principe d'héritage allégerait les documents.

Bibliographie.

[AJ87] Marshall D. Abrams, Albert B. Jeng. Network Security: Protocol Reference Model and the Trusted Computer System Evaluation Criteria. IEEE Network Magasine, April 1987, Vol 1, No.2.

[Br87] Dennis K. Branstad. Considerations for the Security in the OSI Architecture. In IEEE Network Magasine, April 1987, Vol 1, No.2.

[DDDPM93] Marc Derroitte, Eric Dubois, Philippe Du Bois, Michël Petit. Towards a Formal Agent-Oriented Requirements Definition of Manifacturating Systems. Submitted to the International Workshop on the Design of Information Systems for Manifacturating DIISM'93, Tokyo (Japan), November 1993.

[DDP93] Eric Dubois, Philippe Du Bois, Michaël Petit. Elicitating and Formalising Requirements for C.I.M. Information Systems. In C. Rolland, F. Bodart, and C. Cauvet, editors, Proc of the 5th Conference on Advanced Information Systems Engineering - CAiSE'93, pages 252-274, Paris (France), June 8-11, 1993. LNCS 685, Springer-Verlag.

[DDR92] Eric Dubois, Philippe Du Bois, André Rifaut. Elicitating, Structuring and Expressing formal requirements composite systems. In P. Loucopoulos, editor, Proc. of the 4th conference on Advanced Information Systems Engineering - CAiSE'92, pages 327-347, Manchester (UK), May 12-15, 1992. LNCS 593, Springer-Verlag.

[DH76] W. Diffie, M.E. Helman. New Directions in Cryptography. IEEE Trans. Inform. Theory, IT-22, No 6, Nov 1976, pages 644-654.

[DH87] Eric Dubois, Jacques Hagelstein. Reasoning on Formal Requirements: a Lift Control System. In Proceedings of the 4th International Workshop on Software Specification and Design, pages 236-241, Monterey (CA), April 3-4, 1987. IEEE, CS Press

[Eg87] Thierry Eggen. Le modèle OSI: vers une maturité opérationnelle. Nouvelles de la Science et des Technologies, Vol. 5, numéro 1-2, janvier-mars 1987, pp 55-56.

[GJ79] M.R. Garey et D.S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman and Company, San Francisco, CA, 1979.

[GR91] P.C. Goldsack, T.W. Rush. Specifying an Electronic Mail System with HP-SL. Hewlett-Packard Journal December 91.

[GVBP90] R. Govaerts, J. Vandewalle, A. Bosselaers et B. Preneel. Fast Software Implementation Of the Data Encryption Standard (DES), rapport annuel ESAT/COSIC, Katholieke Universiteit Leuven, Belgique, 1990.

[He87] Martin E. Hellman. Commercial Encryption. IEEE Network Magasine, April 1987, Vol 1, No.2.

[Ju87] Robert R. Jueneman. Electronic Document Authentification. IEEE Network Magasine, April 1987, Vol 1, No.2.

[Ma92] Carl-Uno Manros. Messagerie X.400: Introduction aux concepts clés de la normalisation. AFNOR, 1992.

[MM82] C.H. Meyer, M.Matyas. Cryptography: a New Dimension in Computer Data Security. John Wiley & Sons, 1982.

[MRW89a] C. Mitchell, D. Rush, M. Walker. CCITT/ISO Standards for Secure Message Handling. IEEE Journal on Selected Areas in Communication , Vol 7, n°4, may 1989.

[MRW89b] C. Mitchell, D. Rush, M. Walker. A Remark on Hash Functions for Message Authentification. Computers & Security , Vol 8, 1989, pp. 55-58.

[Neu87] G. Neufeld. The EAN Distributed Message System User's Manual (Version 2.1). University of British Columbia (Vancouver, Canada), 1987.

[NOP87] David B. Newman, Jr. Jim K. Omura, Raymond L. Pickholtz. Public Key Management for Network Security. IEEE Network Magasine, April 1987, Vol 1, No.2.

[PST88] C. Pomerance, J.W. Smith et R. Tuler. A Pipe-Line Architecture for Factoring Large Integers with the Quadratic Sieve Algorithm. SIAM Journal On Computing, vol. 17, 1988, pp. 387-403.

[RH93] Jean Ramaekers et Joël Hubin. Sécurité et fiabilité des systèmes informatiques. Syllabus de cours à (3ème licence et maîtrise en informatique). FUNDP, 93

[Ro91] Yves Roggeman. Cryptographie et sécurité informatique. Nouvelles de la Science et des Technologies, Vol. 9, numéro 4, 1991, pp 51-58.

[RSA78] R. Rivest, A. Shamir, L. Adelman. A method for Obtaining Digital Signatures and Public-Key Cryptosystems. CACM, Vol. 21, No 2, Feb 1978, pages 120-128.

[Ru92] Selwyn Russel. Using Public Domain Multiprecision Arithmetic Packages for Computer Security Software Applications on a Personnal Computer. ACM Sigsmall/PC Notes, vol. 18, No 122, Spring/summer 1992.

[RV89] X. Rutten, J. Vanthournout. Etude du logiciel de courrier électronique EAN, mémoire de maîtrise en informatique, FUNDP, 1989.

[SBV90] M. Shand, P. Bertin et J. Vuillemin. Hardware Speedups in Long Integer Multiplication. Proccedings of 2nd ACM Symposium on Parallel Algorithms and Architectures, juillet 1990, pp. 138-145.

[Sou92] B. Soulas. Vers une qualification intégrée des systèmes. RGE, n°10, Nov. 1992.

[WRM93] S. Wu, J. Ramaekers et S. Muftic. An Architecture for a Generalized Secure Message Handling System. Submitted to Hawaï International Conference on System Sciences (HICSS)", january 4-7, 1994.

[Wu92] S. Wu. MHS Security - A Concise Survey. Comp. Net. & ISDN Systems, Vol. 22, Nov. 1992.

[X.400-84] CCITT. Data Communication Networks - Message Handling Systems. Recommendations X.400 - X.430, Malaga - Torremolinos, 1984.

[X.400-88] CCITT. Data Communication Networks - Message Handling Systems. Recommendations X.400 - X.420, Melbourne, 1988.

[X.500] CCITT. The Directory - Overview of Concept, Models and Services. Recommendations X.500, Melbourne, 1988.

[X.509] CCITT. The Directory - Authentication Framework. Recommendations X.509, Melbourne, 1988.

Annexe 1.

Cette annexe reprend les différents types abstraits de données utilisés dans ce travail, ainsi que certaines opérations prédéfinies.

1. TYPES DE BASE.

1.1. BOOLEAN

* true constant *
True: $\rightarrow$ *BOOLEAN*
* false constant *
False: $\rightarrow$ *BOOLEAN*

1.2. INTEGER

* addition *
# + #: *INTEGER* $\times$ *INTEGER* $\rightarrow$ *INTEGER*
* substraction *
# - #: *INTEGER* $\times$ *INTEGER* $\rightarrow$ *INTEGER*
* multiplication *
# * #: *INTEGER* $\times$ *INTEGER* $\rightarrow$ *INTEGER*
* division *
# / #: *INTEGER* $\times$ *INTEGER* $\rightarrow$ *INTEGER*
* modulo *
# mod #: *INTEGER* $\times$ *INTEGER* $\rightarrow$ *INTEGER*
* is a odd number ? *
Odd?: INTEGER $\rightarrow$ *BOOLEAN*
* is an even number ? *
Even?: INTEGER $\rightarrow$ *BOOLEAN*
* is-divisible-by ? *
# *Is divisible by* #: *INTEGER* $\times$ *INTEGER* $\rightarrow$ *BOOLEAN*
* is equal to ? *
# = #: *INTEGER* $\times$ *INTEGER* $\rightarrow$ *BOOLEAN*
* is not equal to ? *
# $\neq$ #: *INTEGER* $\times$ *INTEGER* $\rightarrow$ *BOOLEAN*
* is lower ? *
# < #: *INTEGER* $\times$ *INTEGER* $\rightarrow$ *BOOLEAN*
* is greater ? *
# > #: *INTEGER* $\times$ *INTEGER* $\rightarrow$ *BOOLEAN*
* is lower or equal ? *
# $\leq$ #: *INTEGER* $\times$ *INTEGER* $\rightarrow$ *BOOLEAN*
* is greater or equal ? *
# $\geq$ #: *INTEGER* $\times$ *INTEGER* $\rightarrow$ *BOOLEAN*

1.3. CHAR

* is equal to ? *

$\# = \#: CHAR \times CHAR \rightarrow BOOLEAN$

* is not equal to ? *

$\# \neq \#: CHAR \times CHAR \rightarrow BOOLEAN$

* is lexically before ? *

$\# < \#: CHAR \times CHAR \rightarrow BOOLEAN$

* is lexically after ? *

$\# > \#: CHAR \times CHAR \rightarrow BOOLEAN$

* is lexically before or equal ? *

$\# \leq \#: CHAR \times CHAR \rightarrow BOOLEAN$

* is lexically after or equal ? *

$\# \geq \#: CHAR \times CHAR \rightarrow BOOLEAN$

1.4. SRING

* concatenation *

$\# + \#: STRING \times STRING \rightarrow STRING$

* length *

$Length: STRING \rightarrow INTEGER$

* extraction of a sub-string *

$Sub\text{-}string: STRING \times INTEGER \times INTEGER \rightarrow STRING$

* is equal to ? *

$\# = \#: STRING \times STRING \rightarrow BOOLEAN$

* is not equal to ? *

$\# \neq \#: STRING \times STRING \rightarrow BOOLEAN$

* is lexically before ? *

$\# < \#: STRING \times STRING \rightarrow BOOLEAN$

*is lexically after ? *

$\# > \#: STRING \times STRING \rightarrow BOOLEAN$

* is lexically before or equal ? *

$\# \leq \#: STRING \times STRING \rightarrow BOOLEAN$

* is lexically after or equal ? *

$\# \geq \#: STRING \times STRING \rightarrow BOOLEAN$

2. CONSTRUCTEURS DE TYPE.

2.1. Set

* empty set *

$\{\}: \rightarrow SET[T]$

* set constructor *

$\{\#, \#, \ldots \#\}: T \times T \times \ldots T \rightarrow SET[T]$

* is member of ? (use EqT)*

$\# \in \#: T \times SET[T] \rightarrow BOOLEAN$

* is empty ? *

$Empty?: SET[T] \rightarrow BOOLEAN$

* cardinality *

$Card: SET[T] \rightarrow INTEGER$

* intersection (use EqT) *

$\# \cap \#: SET[T] \times SET[T] \rightarrow SET[T]$

* union *
$\# \cup \# : SET[T] \times SET[T] \rightarrow SET[T]$
* substraction (use EqT) *
$\# \setminus \# : SET[T] \times SET[T] \rightarrow SET[T]$
* adding an element *
$Add : SET[T] \times T \rightarrow SET[T]$
* removing an element (use EqT) *
$Remove : SET[T] \times T \rightarrow SET[T]$
* is equal to ? (use EqT) *
$\# = \# : SET[T] \times SET[T] \rightarrow BOOLEAN$
* is not equal to ? (use EqT) *
$\# \neq \# : SET[T] \times SET[T] \rightarrow BOOLEAN$
* is included ? (use EqT) *
$\# \subset \# : SET[T] \times SET[T] \rightarrow BOOLEAN$

2.2. Le Produit Cartésien (Cartesian Product).

* cartesian product constructor *
$<\#, \#,... \#>: T1 \times T2 \times ... Tn \rightarrow CP[T1, T2, ... Tn]$
* is equal to ? (use EqTi) *
$\# = \#: CP[T1, T2, ... Tn] \times CP[T1, T2, ... Tn] \rightarrow BOOLEAN$
* is not equal to ? (use EqTi) *
$\# \neq \#: CP[T1, T2, ... Tn] \times CP[T1, T2, ... Tn] \rightarrow BOOLEAN$

2.3. Le type UNION.

* is equal to ? (use EqTi on Ti) *
$\# = \#: UNION[T1, T2, ... Tn] \times UNION[T1, T2, ... Tn] \rightarrow BOOLEAN$
* is not equal to ? (use EqTi on Ti) *
$\# \neq \#: UNION[T1, T2, ... Tn] \times UNION[T1, T2, ... Tn] \rightarrow BOOLEAN$
* test for type membership *
$Is\text{-}of\text{-}T: UNION[T1, T2, ... Tn] \times T \rightarrow BOOLEAN$

2.4. La suite (the Sequence).

* empty sequence *
$[]: \rightarrow SEQ[T]$
* sequence constructor *
$[\#, \#,... \#]: T \times T \times ... T \rightarrow SEQ[T]$
* is member of ? (use EqT) *
$\# \in \#: T \times SEQ[T] \rightarrow BOOLEAN$
* is empty ? *
$Empty: SEQ[T] \rightarrow BOOLEAN$
* length *
$Length: SEQ[T] \rightarrow INTEGER$
* first element *
$First: SEQ[T] \rightarrow T$
* last element *
$Last: SEQ[T] \rightarrow T$
* i^{th} element *
$Ith: SEQ[T] \times INTEGER \rightarrow T$

* part of a sequence *

Sub-seq: SEQ[T] × *INTEGER* × *INTEGER* → *SEQ[T]*

* concatenation *

 # + # : SEQ[T] × *SEQ[T]* → *SEQ[T]*

* adding an element at end *

Append: SEQ[T] × *T* → *SEQ[T]*

* inserting an element at the i^{th} position *

Add-ith: SEQ[T] × *T* × *INTEGER* → *SEQ[T]*

* removing the i^{th} element *

Remove-ith: SEQ[T] × *INTEGER* → *SEQ[T]*

* is equal to ? (use EqT) *

 # = # : SEQ[T] × *SEQ[T]* → *BOOLEAN*

1.2.5. Le tableau (the table).

* is in range ? (use EqT) *

 # ∈ # : T × *TBL(ID,T)* → *BOOLEAN*

* is in domain ? (use EqID) *

In-dom: ID × *TBL(ID,T)* → *BOOLEAN*

* is empty ? *

Empty?: TBL(ID,T) → *BOOLEAN*

* acces to an element (use EqID) *

 # [#] [EqID]: TBL(ID,T) × *ID* → *T*

* insertion (use EqID) *

Insert: TBL(ID,T) × *ID* × *T* → *TBL(ID,T)*

* suppression (use EqID) *

Delete: TBL(ID,T) × *ID* → *TBL(ID,T)*

* modification (use EqID) *

Modify: TBL(ID,T) × *ID* × *T* → *TBL(ID,T)*

* is equal to ? (use EqT and EqID) *

 # = #: TBL(ID,T) × *TBL(ID,T)* → *BOOLEAN*

* is not equal to ? (use EqT and EqID) *

 # ≠ #: TBL(ID,T) × *TBL(ID,T)* → *BOOLEAN*

Annexe 2.

Cette annexe reprend le vocabulaire des différents agents utilisés au chapitre III. Pour chaque agent, un tableau reprend pour chaque action et chaque composante d'état sous sa responsabilité:

- le nom de l'action ou de la composante d'état,
- les paramètres ou attributs utilisés par cette action ou cette composante d'état,
- les agents pour lesquels cette action ou cette composante d'état est exportée,
- une descrition de cette action ou de cette composante d'état.

Agent USER

Action ou composante d'état	paramètres	exportée vers	description
Change-folder-req	UA, MESS-FOLDER	UA	Action permettant à l'agent User de demander à l'agent UA qu'il change de folder courant (dossier).
Get-ean-session-req	UA, SESS-PARAM	UA	Action permettant à l'agent User de demander une session avec EAN.
Message-compose-req	UA	UA	Action permettant à l'agent User de demander à l'agent UA de se mettre en mode de composition de message.
Message-delete-req	UA, INTEGER	UA	Action permettant à l'agent User de demander à l'agent UA la suppression d'un message dans un folder courant.
Message-read-req	UA, INTEGER	UA	Action permettant à l'agent User de demander à l'agent UA la lecture d'un message.
Message-submission-req	UA	UA	Action permettant à l'agent User de demander à l'agent UA la soumission du message en cours de manipulation (compose, lecture).
Message-wait-req	UA, INTEGER	UA	Action permettant à l'agent User de demander à l'agent UA la sauvegarde du message qu'il manipulait sans le soumettre au MTA.
Stop-ean-session-req	UA	UA	Action permettant à l'agent User de stopper une session avec EAN.
UA-accessible			Instance de type UA contenant l'UA accessible par un utilisateur.
UA-already-used			Instance de type BOOL indiquant si une session EAN est active pour un agent User.
UA-drop-req	UA	UA	Action permettant à à l'agent User de demander à l'agent UA qu'il utilise les services du DS afin que celui-ci supprime les informations concernant l'agent UA.
UA-find-req	UA, UA-INFO	UA	Action permettant à l'agent User de demander à l'agent UA qu'il utilise les services du DS afin que celui-ci recherche les informations concernant un UA
UA-register-req	UA, UA-INFO	UA	Action permettant à l'agent User de demander à l'agent UA qu'il utilise les services du DS afin que celui-ci stocke les informations concernant un UA.

Tableau 3.1.: Description du vocabulaire des informations sous la responsabilité de l'agent User

Agent UA

Action ou composante d'état	paramètres	exportée vers	description
Action-in-work			Instance de type booléen précisant si l'agent User a formulé une requête auprès de l'agent UA.
Change-folder-conf	USER,MESS -FOLDER	User	Action permettant à l'agent UA d'avertir l'agent User que le nouveau folder est le folder courant.
Draft-folder			Instance contenant le folder courant.
Draft-message		User	Instance contenant le message en cours de manipulation.
Draft-pos			Instance précisant l'endroit du Stored-messages d'où provient le message actuellement dans Draft-message.
DS-DROP-Req		DS	Action permettant à l'agent UA de demander à l'agent DS qu'il détruise les informations le concernant.
DS-FIND-Req	UA-INFO	DS	Action permettant à l'agent UA de demander à l'agent DS qu'il recherche les informations concernant un UA
DS-REG-Req	UA-INFO	DS	Action permettant à l'agent UA de demander à l'agent DS qu'il registre les informations le concernant.
Get-ean-session-conf	USER,BOOL	User	Action permettant à l'agent UA de confirmer ou d'infirmer la demande de connexion de l'agent User avec EAN.
Message-compose-conf	USER,BOOL	User	Action permettant à l'agent UA de confirmer ou d'infirmer la demande de composition de message de l'agent User.
Message-delete-conf	USER,INTE GER, BOOL	User	Action permettant à l'agent UA de confirmer ou d'infirmer la demande de suppression de message de l'agent User.
Message-folders		User	Population composée d'une table contenant, pour tous les folders d'un UA, une table d'identifiants de messages.
Message-read-conf	USER, INTEGER, BOOL	User	Action permettant à l'agent UA de confirmer ou d'infirmer la demande de lecture de message de l'agent User.
Message-submission-conf	USER,BOOL	User	Action permettant à l'agent UA de confirmer ou d'infirmer la demande de soumission de message de l'agent User.
Message-wait-conf	USER, INTEGER, BOOL	User	Action permettant à l'agent UA de confirmer ou d'infirmer la demande de sauvegarde sans soumission de message de l'agent User.
MTA-accessible			Instance contenant l'information sur l'agent MTA accessible par l'agent UA.
Stop-ean-session-conf	USER,BOOL	User	Action permettant à l'agent UA de confirmer ou d'infirmer la demande de l'agent User consistant à clore la session avec EAN.
Stored-messages		User	Population composée d'une table contenant, pour tous les identifiants de messages, les messages proprement dits.
Stored-reports		User	Population composée d'une table contenant, pour tou les identifiants, de messages les rapports concernant ces messages.
UA-drop-conf	USER	User	Action permettant à l'agent UA d'avertir l'agent User que les informations le concernant ont été supprimées du DS.
UA-find-conf	USER, UA-INFO	User	Action permettant à l'agent UA de renvoyer le résultat d'une requête de recherche formulée au DS à l'agent User.
UA-header		User	Population composée d'une table contenant, pour tous les identifiants de messages, les en-têtes de ceux-ci.
UA-register-conf	USER, UA-INFO	User	Action permettant à l'agent UA d'avertir l'agent User que les informations le concernant ont été enregistrées par le DS.
UA-stat			Instance contenant le status de l'UA (IDLE, NOT IDLE).
User-in-session			Instance contenant l'agent User en cours de session.

Tableau 3.2.: Description du vocabulaire des informations sous la responsabilité de l'agent UA

Agent MTA

Action ou composante d'état	paramètres	exportée vers	description
UA-knows			Population contenant l'ensemble des agents UA connus par le MTA contenant cette population.
MTA-knows			Population contenant l'ensemble des agents MTA connus par le MTA contenant cette population.
LOGON-Conf	UA	UA	Action. cf. tableau 1.2 chapitre I
LOGOFF-Conf	UA	UA	Action. cf. tableau 1.2 chapitre I
DELIVER-Ind	UA, MESSAGE	UA	Action. cf. tableau 1.2 chapitre I
NOTIFY-Ind	UA, REPORT, MESSID	UA	Action. cf. tableau 1.2 chapitre I
MTA-messages			Population composée d'une table contenant, pour chaque message, un ensemble de destinations auxquelles le message devra être délivré ou transféré.
MTA-reports			Population composée d'une table contenant, pour chaque rapport, la destination à laquelle celui-i devra être notifié ou transféré.
Message-transfert	MTA, MESSAGE	MTA	Action permettant à l'agent MTA de transférer vers un autre agent MTA le message donné en paramètre
Report-transfert	REPORT, DEST	MTA	Action permettant à l'agent MTA de transférer vers un autre agent MTA le rapport de livraison/nonlivraison donné en paramètre
Report-generate	UA, REPORT, MESSID		Action permettant à l'agent MTA de générer le rapport de livraison/nonlivraison donné en paramètre
Message-routing	UA, MESSAGE, DEST		Action permettant à l'agent MTA de déterminer dans entité sera transférer le message (UA si livraison, MTA si transfert)
SUBMIT-Conf	UA, MESSAGE		Action. cf. tableau 1.2 chapitre I

Tableau 3.3.: Description du vocabulaire des informations sous la responsabilité de l'agent MTA

Agent DS

DS-uas	Population composée d'une table contenant les UAs connus du DS.
UA-drop-conf(ua)	Action permettant à l'UA d'informer l'agent User que sa requête de suppression des informations concernant un UA contenues dans le DS s'est déroulée correctement.
UA-find-conf(ua,ua-inf)	Action permettant à l'UA d'informer l'agent User que sa requête de recherche d'informations concernant un UA s'est déroulée correctement.
UA-register-conf(ua,ua-inf)	Action permettant à l'UA d'informer l'agent User que sa requête d'enregistrement des informations concernant un UA s'est déroulée correctement.

Tableau 3.4.: Description du vocabulaire des informations sous la responsabilité de l'agent UA

Annexe 3.

Cette annexe reprend les différents types abstraits de données utilisés au chapitre III dans la phase de spécification des besoins de EAN, ainsi que certaines opérations prédéfinies.

type MESSAGE*
CP[Content CONTENT,
Enveloppe ENVELOPPE]

Type CONTENT*

Type ENVELOPPE*
CP[Origin UA,
Types TYPES,
Mess-id MESS-ID,
Defer DEFER,
Import IMPORT,
Perms PERMS,
Recip UAS,
Cont-type CONT-TYPE]

Type REPORT*

Type UA-HEADER*
CP [Status CHAR,
Origin UA
Recip UAS,
In-out STRING,
Date DATE,
Subject STRING]

Type STATUS*
CP [New BOOL,
Read BOOL,
Draft BOOL,
Sent BOOL,
Rprt BOOL,
Rply BOOL,
Fwrd BOOL,
Del BOOL]

Type UA-INFO*
CP[Nun STRING,
Alternates ALTERNATES,
Nua STRING,
Adress STRING,
Phone STRING,
Description STRING,

Type UA-STATUS*
value in {IDLE, NOT IDLE}

Type AUTH-PARAM*

Type SESS-PARAM*

Type TMESS

TBL[INTEGER → STRING]

opération sur le type TMESS

Max: TMESS → INTEGER

Annexe 4.

Cette annexe reprend le vocabulaire des différents agents utilisés au chapitre III. Pour chaque agent, un tableau reprend pour chaque action et chaque composante d'état sous sa responsabilité:

- le nom de l'action ou de la composante d'état,
- les paramètres ou attributs utilisés par cette action ou cette composante d'état,
- les agents pour lesquels cette action ou cette composante d'état est exportée,
- une descrition de cette action ou de cette composante d'état.

Agent USER

Action ou composant d'état	paramètres	exportée vers	description
Alg-key-drop-req	UA-ALG-KEY	UA	Action permettant à l'agent User de demander à l'agent UA qu'il util les services du DS afin que celui-ci supprime la structure ALG-K donnée en paramètre, structure qui contient un identifiant d'algorith de chiffrement utile à la vérification de la signature ainsi que la publique pour cet algorithme.
Alg-key-find-req	UA-ALG-KEY ALG-KEY	UA	Action permettant à l'agent User de demander à l'agent UA qu'il util les services du DS afin que celui-ci recherche la structure ALG-K correspondant à un critère de recherche.
Alg-key-put-req	UA-ALG-KEY ALG-KEY	UA	Action permettant à l'agent User de demander à l'agent UA qu'il util les services du DS afin que celui-ci insère la structure ALG-K donnée en paramètre.
Alg-key-modify-req	UA-ALG-KEY	UA	Action permettant à l'agent User de demander à l'agent UA qu'il util les services du DS afin que celui-ci modifie la structure ALG-KEY fonctiuon d'un critère de recherche.
Get-security-states-req	UA,SEC-STATES	UA	Action permettant à l'agent User de spécifier les services de sécur qu'il désire adjoindre à un message lorsque celui-ci sera soumis.
Vrfy-security-states-rec	UA,SEC-STATES	UA	Action permettant à l'agent User de demander au serveur de sécur de vérifier le respect en sécurité d'un message sécurisé.
Identification-conf	UA,AUTH-PARAM	UA	Action permettant à l'agent User de répondre à la deman d'identification de l'UA.
UA-key-conf	UA,UA, ALG-KEY		Action permettant à l'agent User de fournir une clé et un identifi d'algorithme à l'UA afin que celui-ci le transmette au serveur sécurité.

Tableau 3.1.: Description du vocabulaire des informations sous la responsabilité de l'agent User

Agent UA

Action	paramètres	exportée vers	description
Security-states		User	Population composée d'une table telle que pour tous les messag (messid), une structure de type SECURITY-STATES y est associ permettant à l'UA de demander des services de sécurité pour soumission ou d'accéder au résultats de la vérification du respect.
Sec-results		User	Population composée d'une table telle que pour tous les messag (messid), une structure contenant les erreurs causées au sein serveur de sécurité.
Draft-sec-states		User	Instance contenant les mêmes informations que Security-states m pour le Draft-message.
Draft-sec-results		User	Instance contenant les mêmes informations que Sec-results mais p le Draft-message.
Identification-req	USER,BOOL	User	Action permettant à l'agent UA de demander à l'agent User s'identifier.
BEGIN-SECREQ-Req	SEC-SERV, MESSAGE, SEC-STATES	Sec-serv	Action permettant à l'UA de demander à l'agent Sec-serv de sécuri un message à soumettre.
BEGIN-SECVRFY-Re	SEC-SERV, MESSAGE, SEC-STATES	Sec-serv	Action permettant à l'UA de demander à l'agent Sec-serv de vérifier la sécurité d'un message.
Vrfy-security-states-co	USER, SEC-STATES	User	Action permettant à l'agent UA de confirmer que le serveur de sécur a fini son service.
Get-security-states-con	USER, SEC-STATES	User	Action permettant à l'agent UA de confirmer que le Draft-message s bien sécurisé lors de sa soumission.
UA-KEY-Conf	SEC-SERV, UA, ALG-KE	Sec-serv	Action permettant à l'agent UA de fournit une clé et un identifiant d'algorithme au serveur de sécurité.
Sec-serv-accessible			Instance contenant la valeur du serveur de sécurité auquel peut accé l'agent UA.
UA-key-req	USER,UA, ALG-KEY	User	Action permettant à l'agent UA de demander une clé et un identifian d'algorithme à l'agent User.
Alg-key-drop-conf	UA- ALG-KEY	User	Action permettant à l'agent UA de confirmer à l'agent UA que le DS supprimé la structure ALG-KEY.
Alg-key-find-conf	UA- ALG-KEY ALG-KEY	User	Action permettant à l'agent UA de fournir le résultat de la recherc d'une clé et d'un identifiant d'algorithme correspondant à un critère recherche et exécutée par l'agent DS.
Alg-key-put-req	UA- ALG-KEY ALG-KEY	User	Action permettant à l'agent UA de confirmer l'insertion d'une clé d'un identifiant d'algorithme par le DS.
Alg-key-modify-req	UA- ALG-KEY	User	Action permettant à l'agent UA de confirmer la modification d'une et d'un identifiant d'algorithme correspondant à un critère de recherc et exécutée par l'agent DS.
ALG-KEY-FIND-Req	UA, ALG-KEY ALG-KEY	DS	Action permettant à l'agent UA de demander à l'agent DS une recherche d'une clé et d'un identifiant d'algorithme correspondant à u critère de recherche.

Tableau 3.2.: Description du vocabulaire des informations sous la responsabilité de l'agent UA

Annexe 5.

Cette annexe reprend les différents types abstraits de données utilisés au chapitre VI dans la phase de spécification du serveur de sécurité. Certains types définis au chapitre III ont été étendus en correspondance aux nouveaux buts.

type MESSAGE
CP[Content CONTENT,
Enveloppe ENVELOPPE]

Type CONTENT
UNION[DECR-CONTENT,
DECRYPTED-DATA]

Type ENVELOPPE
CP[Origin UA,
Types TYPES,
Mess-id MESS-ID,
Defer DEFER,
Import IMPORT,
Perms PERMS,
Recip UAS,
Cont-type CONT-TYPE,
Message-tokens MESS-TOKENS]

Type MESS-TOKENS
SET[Message-token MESS-TOKEN]

opération sur le type MESS-TOKENS
Get-env-mt: MESS-TOKENS → MESS-TOKEN

Type MESS-TOKEN
CP[Signature-alg-id ALGORITHM-ID,
Recip-name UAS,
Time UTCTIME,
Signed-data SIGNED-DATA,
Encryption-alg-id ALGORITHM-ID,
Encrypted-data MT-DATA]
Mt-signature ENCRYPTED-DATA]

opération sur le type MESS-TOKEN
Get-mt-st: MESS-TOKEN → SEC-STATES

Type MT-DATA
UNION[DECRMT-DATA,
ENCRMT-DATA]

Type SIGNED-DATA
CP[Content-integrity-alg-id ALGID,
Content-integrity-check ENCRYPTED-DATA,
Message-security-label SEC-LABEL,
Proof-of-delivery-req PROOF-DEL-REQ,
Message-sequence-number INTEGER]

Type DECRYPTED-DATA

UNION[DECR-CONTENT,
DECRMT-DATA,
BITSTRING]

opération sur le type DECRYPTED-DATA

Hash: DECRYPTED-DATA X ALG-KEY → BITSTRING

Sym-encrypt: DECRYPTED-DATA X ALG-KEY → ENCRYPTED-DATA

Asym-encrypt: DECRYPTED-DATA X ALG-KEY → ENCRYPTED-DATA

Type DECRMT-DATA

CP[Content-confidentiality-key BITSTRING,
Content-integrity-check ENCRYPTED-DATA,
Message-security-label SEC-LABEL,
Content-integrity-key CONT-INT-KEY,
Message-sequence-number INTEGER]

Type ENCRYPTED-DATA

opération sur le type ENCRYPTED-DATA

Sym-decrypt: ENCRYPTED-DATA X ALG-KEY → DECRYPTED-DATA

Asym-decrypt: ENCRYPTED-DATA X ALG-KEY → DECRYPTED-DATA

Type SEC-STATES

CP[Sec-req-states SECREQSTATES,
Sec-vrfy-states SECVRFYSTATES]

Type SECREQSTATES

CP[Proof-of-submission SECREQBOOL,
Proof-of-delivery SECREQBOOL,
Content-integrity SECREQBOOL,
Content-confidentiality CCREQBOOL]

Type CCREQBOOL

TBL[Ua UA → Secreqbool: SECREQBOOL]

Type SECVRFYSTATES

UNION[OR-VRFY-ST,
RECIP-VRFY-ST]

Type OR-VRFY-ST

CP[Proof-of-submission SECVRFYBOOL,
Proof-of-delivery SECVRFYBOOL],

Type RECIP-VRFY-ST

CP[Content-integrity SECVRFYBOOL,
Content-confidentiality SECVRFYBOOL]]

Type SECREQBOOL

value in {REQUESTED, NOTREQUESTED}

Type SECVRFYBOOL

value in {TRUE, FALSE, UNDEFINED, INACCESSIBLE}

Type SEC-RESULTS

CP[Sec-req-result SECREQRESULTS,
Sec-vrfy-result SECVRFYRESULTS]

Type SECREQRESULTS
CP[Mtok-result MTSERVERR,
Pr-of-subm-result SECSERVERR,
Pr-of-deliv-result SECSERVERR,
Cont-int-result SECSERVERR,
Cont-conf-result CCSERVERR]

Type CCSERVERR
TBL[UA → SECSERVERR]

Type MTSERVERR
TBL[MESS-TOKEN → SECSERVERR]

Type SECVRFYRESULT
UNION[OR-VRFY-RES
RECIP-VRFY-RES]

Type OR-VRFY-RES
CP[Pr-of-subm-result SECSERVERR,
Pr-of-deliv-result SECSERVERR,
Mtok-result MTSERVERR]

Type RECIP-VRFY-RES
CP[Cont-int-result SECSERVERR,
Cont-conf-result SECSERVERR,
Mtok-result MTSERVERR]]

Type SECSERVERR
CP[Seq-req-err-no INT,
Seq-req-err-descr STRING]

Type UA-HEADER
CP [Status CHAR,
Origin UA
Recip UAS,
In-out STRING,
Date DATE,
Subject STRING]

Type STATUS
CP [New BOOL,
Read BOOL,
Draft BOOL,
Sent BOOL,
Rprt BOOL,
Rply BOOL,
Fwrd BOOL,
Del BOOL]

Type UA-INFO
CP[Nun STRING,
Alternates ALTERNATES,
Nua STRING,
Adress STRING,
Phone STRING,
Description STRING,

Type UA-STATUS
value in {IDLE, NOT IDLE}

Type AUTH-PARAM

Type SESS-PARAM

Type ALG-KEY

Type SEC-INFO
CP[Processus PROCESSUS,
Current-ss SET[Sec-id STRING],
SS-done SET[Sec-id STRING],
SS-to-be-done SET[Sec-id STRING]]

Type PROCESSUS
CP[Processus-id STRING,
Sec-id STRING]

Type TMESS
TBL[Int INTEGER → Messid STRING]

opération sur le type TMESS
Max: TMESS → INTEGER